
AMIGA OS 3.1

COPYRIGHT

HAFTUNGSAUSSCHLUSS

WARENZEICHEN

Coverdesign und Druck im Selbstverlag von Village Tronic
Village Tronic Marketing GmbH, Wellweg 95, 31157 Sarstedt, Germany

Das vorliegende Buch wurde von Isabelle Vesey und Robert Stephenson Weir unter Benutzung verschiedener Commodore-Rechnersysteme erstellt.

Inhaltsverzeichnis

1. Einführung in ARexx

2. Erste Schritte

3. Elemente der Programmiersprache ARexx

4. Befehle

5. Funktionen

6. Fehlersuche

7. Syntaxanalyse

Anhang A
Fehlermeldungen

Anhang B
Kommandohilfsprogramme

Glossar

Index

Willkommen bei ARexx

ARexx, das Amiga-Gegenstück zur IBM-Programmiersprache REXX, ermöglicht Ihnen die individuelle Gestaltung Ihrer Arbeitsumgebung. ARexx eignet sich besonders gut für Skripts, über die Anwendungen gesteuert und modifiziert sowie deren Interaktion mit anderen Anwendungen definiert werden.

Das vorliegende Handbuch bietet eine Einführung in ARexx, beschreibt das Erstellen von ARexx-Programmen und enthält Referenzinformationen zu ARexx-Kommandos.

Kapitel 1. Einführung in ARexx: Dieses Kapitel bietet einen Überblick über ARexx, die Funktionsweise der Programmiersprache auf einem Amiga und die wichtigsten Funktionen der Programmiersprache.

Kapitel 2. Erste Schritte: Hier wird beschrieben, wo ARexx-Programme zu speichern sind und wie ein ARexx-Programm aufgerufen wird. Außerdem werden einige Programmierbeispiele vorgestellt.

Kapitel 3. Elemente von ARexx: Dieses Kapitel beschreibt detailliert die Regeln und Konzepte, auf denen die Programmiersprache ARexx basiert.

Kapitel 4. Befehle: Dieses Kapitel enthält eine alphabetische Liste der ARexx-Befehle. Darunter versteht man Sprachanweisungen zur Steuerung von Aktionen.

Kapitel 5. Funktionen: Dieses Kapitel beschreibt die Verwendung von Funktionen (von ARexx verwendete Programmanweisungen) und enthält eine alphabetische Liste der integrierten ARexx-Funktionen.

Kapitel 6. Testhilfe: Dieses Kapitel beschreibt die Funktionen der Fehlersuchhilfe auf Quelltextebene (source-level debugging), die bei der Entwicklung und beim Testen von Programmen eingesetzt werden können.

Kapitel 7. Syntaxanalyse: Dieses Kapitel beschreibt, wie Informationsmuster aus Zeichenfolgen extrahiert werden können.

Anhang A. Fehlermeldungen: Dieser Anhang enthält eine Liste der ARexx-Fehlermeldungen.

Anhang B. Kommandohilfsprogramme: Dieser Anhang enthält eine Liste der ARexx-Kommandos, die von der Shell aus aufgerufen werden können.

Glossar. Das Glossar enthält gebräuchliche ARexx-Begriffe und die dazugehörigen Definitionen.

Dokumentkonventionen

Im vorliegenden Handbuch gelten folgende Konventionen:

SCHLÜSSELWÖRTER	Schlüsselwörter erscheinen stets in Großbuchstaben; bei den Argumenten wird allerdings Groß- und Kleinschreibung verwendet.
ausdruck	Erforderliche Argumente erscheinen in Kleinbuchstaben.
| (senkrechter Strich)	Alternative Auswahlmöglichkeiten werden durch senkrechte Striche voneinander getrennt.
{ } (geschweifte Klammern)	Erforderliche Alternativen stehen in geschweiften Klammern.
[ ] (eckige Klammern)	Wahlfreie Bestandteile eines Befehls stehen in eckigen Klammern.

<n>	Variablen stehen in spitzen Klammern. Diese spitzen Klammern dürfen bei Eingabe von Variablen nicht mit eingegeben werden.
`Courier`	Text in der Schriftart Courier stellt Ausgabe von ARexx-Programmen oder andere auf dem Monitor angezeigte Informationen dar.
Taste1 - Taste2	Tastenkombinationen mit Bindestrich (-) bedeuten das gleichzeitige Drücken der jeweiligen Tasten.
Taste1, Taste2	Tastenkombinationen mit Komma (,) als Trennzeichen bedeuten, daß die Tasten nacheinander zu drücken sind.
Amiga-Tasten	Zwei Tasten auf der Amiga-Tastatur dienen zur Ausführung von Sonderfunktionen. Die linke Amiga-Sondertaste befindet sich links von der Leertaste und trägt ein großes, schwarz ausgefülltes A. Die rechte Amiga-Sondertaste befindet sich rechts von der Leertaste und trägt ein großes, im Umriß gezeichnetes A.

Zusätzliche Informationsquellen

Die folgenden Bücher enthalten weitere Informationen zum Erlernen und Verwenden von ARexx:

Modern Programming Using REXX, by R.P. O'Hara and D.G. Gomberg, Prentice-Hall, 1985

The REXX Language: A Practical Approach to Programming, by M. F. Cowlishaw, Prentice-Hall, 1985. Auch in Deutsch verfügbar als "Die Programmiersprache REXX".

Programming in REXX, J. Ranade IBM Series, by Charles Daney.

Using ARexx on the Amiga, by Chris Zamara and Nick Sullivan, Abacus, 1991.

Amiga ROM Kernel Reference Manual: Libraries, Third Edition, Addison-Wesley, 1992.

Kapitel 1
Einführung in ARexx

Die Programmiersprache ARexx dient als zentraler Knotenpunkt, über den Anwendungen — auch solche anderer Hersteller — Daten und Kommandos untereinander austauschen können. Mit ARexx ist es beispielsweise möglich, ein Telekommunikationspaket so zusammenzustellen, daß es einen elektronischen Briefkasten ("Mailbox") anwählt, die gewünschten finanztechnischen Daten dort abruft und speichert und diese dann automatisch, d. h. ohne Eingreifen des Benutzers, in ein separates Tabellenkalkulationsprogramm überträgt, um sie dort auszuwerten.

ARexx ist eine interpretierende Sprache, die ASCII-Dateien als Eingabe verwendet. Der ARexx-Interpreter ist das Programm RexxMast. Dieses befindet sich in der Schublade "System" der Workbench. RexxMast überwacht die Ausführung eines ARexx-Programms. Wenn RexxMast beim Übersetzen oder Ausführen einer Zeile auf einen Fehler stößt, wird der Programmablauf unterbrochen und eine Fehlermeldung angezeigt. Dieses interaktive Testen ist einerseits eine Lernhilfe, andererseits erleichtert es die Programmerstellung bzw. -korrektur, da die Fehlermeldung unmittelbar mitteilt, an welcher Stelle der Fehler aufgetreten ist.

1.1 An welche Zielgruppe wendet sich ARexx?

ARexx-Programme und -Skripts setzen keine detaillierten Kenntnisse über den Amiga voraus, allerdings sollten Sie über Grundkenntnisse über folgende Punkte verfügen:

- Öffnen einer Shell und Eingeben von AmigaDOS-Befehlen
- Verwenden eines Texteditors, z. B. ED oder MEmacs
- Erstellen einer Benutzer-Startdatei

Bevor Sie daran gehen können, Skripts zu verändern oder eigene ARexx-Skripts zu erstellen, sollten Sie die Grundlagen der Arbeitsumgebungen der Amiga-Workbench und von AmigaDOS kennen. Erfahrene Amiga-Benutzer werden feststellen, daß es sich mit ARexx leichter und effizienter arbeiten läßt als mit AmigaDOS. ARexx kann auch eingesetzt werden, um bereits bestehende AmigaDOS-Befehle und -Skripts zu erweitern oder zu ersetzen sowie integrierte Anwendungen zu erstellen.

1.2 ARexx auf dem Amiga

ARexx wird auf allen Amiga-Hardwarekonfigurationen unterstützt. Seit der Freigabe der Amiga Workbench Version 2.0 ist ARexx integraler Bestandteil des Amiga-Betriebssystems. ARexx verwendet insbesondere zwei wichtige Funktionen des Amiga-Betriebssystems: Multitasking und Prozeßkommunikation.

Multitasking ist die Fähigkeit, gleichzeitig mehrere Programme auszuführen. Sie können z. B. zur selben Zeit eine Datei edieren, eine Diskette formatieren und die Farben Ihres Monitors neu einstellen.

Unter Prozeßkommunikation (engl. Interprocess Communication - IPC) versteht man die Fähigkeit eines Computers, Informationen zwischen den gerade laufenden Anwendungen auszutauschen. Die Prozeßkommunikation wird über sogenannte Message-Ports (dt. etwa Nachrichtenschnittstellen) abgewickelt. Ein Message-Port ist eine in einem Anwendungsprogramm enthaltene Adresse, über die Nachrichten und Kommandos gesendet und empfangen werden können. Jeder Message-Port trägt einen Namen, und die Übertragung einer Nachricht an eine Anwendung erfordert die Angabe des jeweiligen Port-Namens in einem ARexx-Skript.

Die Operationen beim Senden und Empfangen von Nachrichten finden in folgender Reihenfolge statt:

1. Bei der Initialisierung öffnet ein Anwendungsprogramm einen Message-Port.
2. Die Anwendung wartet auf den Empfang einer Nachricht.
3. Das Amiga-Betriebssystem teilt der Anwendung mit, daß eine Nachricht eingetroffen ist.
4. Die Anwendung führt eine der Nachricht entsprechende Aktion aus.
5. Die Anwendung teilt dem Absender der Nachricht (ARexx) mit, daß die Nachricht empfangen und verarbeitet wurde.

Diese Art der Nachrichtenübertragung ist nicht auf eine Anwendung und ARexx begrenzt. Mehrere Anwendungen können Nachrichten über ARexx als zentrale Übertragungsstation senden, empfangen und austauschen. Natürlich müssen alle beteiligten Anwendungsprogramme zu ARexx kompatibel sein.

1.3 Merkmale von ARexx

Die Programmiersprache ARexx zeichnet sich durch folgende Merkmale aus:

- Typenlose Daten — Daten werden als individuelle Zeichenfolgen behandelt, Variablenwerte werden nicht deklariert.
- Interpretierte Ausführung — Die Fähigkeit von ARexx, ein Programm direkt zu lesen und auszuführen, macht ein separates Kompilieren des Programms überflüssig.
- Automatische Betriebsmittelverwaltung — Dank der automatischen internen Speicherreservierung werden nicht mehr benötigte Zeichenfolgen und Daten entfernt.
- Ablauf- und Ereignisverfolgung und Fehlersuche — Die Ablaufverfolgung bzw. Ereignisverfolgung ermöglicht eine spezielle Fehlerbehandlung, wo ansonsten ein Programmabbruch die Folge wäre. Die Testhilfefunktionen ermöglichen die Überprüfung des gesamten Programms, was den Zeitaufwand für Entwicklung und Test reduziert.
- Funktionsbibliotheken — Externe Funktionsbibliotheken enthal-ten erweiterte, vorprogrammierte Funktionen.

Kapitel 2
Erste Schritte

In diesem Kapitel werden folgende Aktionen beschrieben:

- Starten von ARexx
- Sichern von Programmen
- Speichern von Programmen
- Verwenden von Beispielprogrammen

2.1 Starten von ARexx

Zum Arbeiten mit ARexx wird zunächst das Programm RexxMast aktiviert. Dies kann automatisch oder manuell erfolgen. Bei jedem Starten oder Stoppen von ARexx wird eine Meldung angezeigt.

2.1.1 Automatisches Starten von ARexx

ARexx können Sie auf zwei Arten automatisch starten: indem Sie das Piktogramm RexxMast in die Schublade WBStartup ziehen oder die Datei S: User-Startup (Benutzer-Startdatei) edieren.

Gehen Sie wie folgt vor, um das RexxMast-Piktogramm in die Schublade WBStartup zu ziehen:

1. Öffnen Sie die System-Schublade.
2. Ziehen Sie das RexxMast-Piktogramm über die Schublade WBStartup.
3. Starten Sie Ihren Amiga neu.

Gehen Sie wie folgt vor, um die Datei S:User-Startup zu edieren:

1. Öffnen Sie einen Texteditor.
2. Laden Sie die Datei S:User-Startup.
3. Fügen Sie dort folgende Zeile ein: REXXMAST>NIL:.
4. Speichern Sie die Datei.
5. Starten Sie Ihren Amiga neu.

2.1.2 Manuelles Starten von ARexx

Es gibt zwei Möglichkeiten, RexxMast manuell zu starten: Doppelklicken auf das RexxMast-Piktogramm oder Starten des Programms von einer Shell aus. Benutzer von reinen Diskettensystemen können Speicherplatz sparen, indem sie ARexx nur bei Bedarf starten.

Gehen Sie wie folgt vor, um RexxMast von der Workbench aus zu starten:

1. Öffnen Sie die System-Schublade.
2. Doppelklicken Sie auf das RexxMast-Piktogramm.

Zum Starten von RexxMast von der Shell aus:

1. Öffnen Sie eine Shell.
2. Geben Sie REXXMAST >NIL: ein und drücken Sie die Eingabetaste.

2.2 Informationen zu ARexx-Programmen

ARexx-Programme werden in der Regel im Verzeichnis REXX: gespeichert (dieses ist normalerweise dem Verzeichnis SYS:S zugewiesen). Es wäre zwar möglich, Programme in beliebigen Verzeichnissen zu speichern; die Speicherung im Verzeichnis REXX: hat aber einige Vorteile:

- Das Programm kann ohne Angabe des vollständigen Pfadnamens aufgerufen werden.
- Alle Ihre ARexx-Programme stehen an der gleichen Stelle.
- Die meisten Anwendungsprogramme greifen bei der Suche nach ARexx-Programmen auf das Verzeichnis REXX: zu.

Ein ARexx-Programm könnte nicht nur an beliebiger Stelle gespeichert, sondern auch beliebig benannt werden. Allerdings erleichtern Sie sich die Programmverwaltung sehr, wenn Sie eine einfache Benennungskonvention beachten: Programme, die von der Shell aus aufgerufen werden, sollten die Namenserweiterung .rexx erhalten. Auf diese Weise lassen sie sich von Dateien unterscheiden, die von anderen Anwendungen aufgerufen werden.

2.2.1 Aufrufen von ARexx-Programmen

Das Kommando RX dient zum Aufrufen eines ARexx-Programms. Wird zum Programmnamen ein vollständiger Pfad angegeben, so wird nur das in diesem Pfad angegebene Verzeichnis nach dem Programm durchsucht. Wenn Sie keinen Pfad angeben, erstreckt sich die Suche auf das aktuelle Verzeichnis und das Verzeichnis REXX:.

Solange die Programme im Verzeichnis REXX: gespeichert sind, ist die Angabe der Erweiterung .rexx zum Programmnamen nicht erforderlich, d. h. die Eingabe von:

```
RX Programm.rexx
```

entspricht der Eingabe von:

```
RX Programm
```

Ein kurzes Programm kann auch direkt in der Kommandozeile eingegeben werden, indem die Programmzeile in doppelte Anführungszeichen gestellt wird. Das folgende Programm sendet z. B. fünf Dateien mit den Namen myfile.1 bis myfile.5 an den Drucker und muß als eine Zeile eingetippt werden:

```
RX "DO i=1 to 5;
ADDRESS command 'copy myfile.' || i 'prt:'; END"
```

Wenn eine Anwendung zu ARexx kompatibel ist, können innerhalb dieser Anwendung ARexx-Programme aufgerufen werden. Dazu wählen Sie einen Menüpunkt aus oder geben bestimmte Kommandooptionen an. Näheres dazu entnehmen Sie bitte der Dokumentation zur jeweiligen Anwendung.

ARexx-Programme können auch von der Workbench aus aufgerufen werden. Zu diesem Zweck erstellen Sie ein Programm- oder Projektpiktogramm für das Programm. Das Kommando RX muß als Standardprogramm für das Piktogramm angegeben werden. In das Informationsfenster des Piktogramms geben Sie folgendes ein:

```
Standardprogramm: SYS:Rexxc/RX
```

Beim Öffnen des Piktogramms startet RX das Programm RexxMast (wenn es nicht bereits läuft). Es führt die dem Piktogramm zugeordnete Datei als ARexx-Programm aus.

ARexx akzeptiert zwei Arten von Merkmalen: Console (zur Angabe eines Fensters) und CMD (zur Angabe einer Kommandozeichenfolge). Diese Merkmale geben Sie in das Informationsfenster des Projektpiktogramms wie folgt ein:

```
Console=CON:0/0/640/200/Example/Close
CMD=rexxprogram
```

2.3 Programmbeispiele

Die folgenden Beispiele zeigen, wie Sie ARexx zur Anzeige von Textzeichenfolgen am Bildschirm, zur Ausführung von Berechnungen und zum Aktivieren der Fehlerprüfroutine einsetzen können.

Programme können über jeden beliebigen Texteditor (z. B. ED oder MEmacs) oder ein Textverarbeitungsprogramm eingegeben werden. Bei Verwendung eines Textverarbeitungsprogramms speichern Sie Ihr Programm bitte als ASCII-Datei. ARexx unterstützt den erweiterten ASCII-Zeichensatz (Å, Æ, ß, Umlaute). Diese erweiterten Zeichen werden als gewöhnliche Druckzeichen erkannt und auch korrekt von Klein- in Großschreibung umgewandelt.

Die Beispiele zeigen auch die Verwendung einiger syntaktischer Erfordernisse von ARexx, z. B:

- Kommentarzeilen
- Regeln für Zeichenabstände
- Unterscheidung zwischen Groß- und Kleinschreibung
- Verwendung einzelner und doppelter Anführungszeichen

Jedes ARexx-Programm besteht mindestens aus einer das Programm beschreibenden Kommentarzeile und einem Befehl, der Text im Konsolenfenster anzeigt. ARexx-Programme müssen stets mit einer Kommentarzeile beginnen. Die einleitende Zeichenfolge /* (zu der stets eine Abschlußzeichenfolge */ vorhanden sein muß) teilt dem RexxMast-Interpreter mit, daß es sich um ein ARexx-Programm handelt. Ohne die Zeichenfolgen /* und */ erkennt RexxMast die Datei nicht als ARexx-Programm. Nach Beginn der Programmausführung ignoriert ARexx alle weiteren Kommentarzeilen in der Datei. Die Kommentarzeilen sind jedoch zum Verstehen des Programms ausgesprochen hilfreich, da sie das Verständnis der Logik und des Aufbaus des Programms wesentlich erleichtern.

2.3.1 Amiga.rexx

Dieses Programm zeigt, wie das Wort SAY in einem ARexx-Befehl zur Anzeige von Textzeichenfolgen am Bildschirm eingesetzt wird.

Befehle sind Sprachanweisungen, die eine bestimmte auszuführende Aktion näher bestimmen. Jede Anweisung beginnt mit einem Symbol, im folgenden Beispiel mit dem Wort SAY. (Symbole werden beim Programmablauf grundsätzlich in Großbuchstaben umgewandelt.) Auf SAY folgt ein Beispiel für eine Zeichenfolge. Unter einer Zeichenfolge versteht man eine Reihe von Zeichen, die in ein Paar von einfachen (Apostroph, Alt-Ä) oder doppelten Anführungszeichen eingeschlossen sind (' oder ").

Programm 1. Amiga.rexx

```
/*Ein einfaches Programm*/
SAY 'Amiga, der Kreativ-Computer.'
```

Geben Sie den obigen Programmtext ein und speichern Sie ihn unter dem Namen REXX:Amiga.rexx. Zum Aufrufen des Programms öffnen Sie nun ein Shell-Fenster und geben folgendes ein:

```
RX Amiga
```

Der vollständige Pfad- und Programmname lautet zwar Rexx:Amiga.rexx, Sie brauchen aber weder den Verzeichnisnamen REXX: noch die Namenserweiterung .rexx einzugeben, wenn Sie das Programm im Verzeichnis REXX: gespeichert haben.

In Ihrem Shell-Fenster sollte nun der folgende Text erscheinen:

```
Amiga, der Kreativ-Computer.
```

2.3.2 Alter.rexx

Dieses Programm zeigt eine Eingabeaufforderung an und liest dann die eingegebenen Daten.

Programm 2. Alter.rexx

```
/*Alter in Tagen berechnen*/
SAY 'Bitte Ihr Alter eingeben:'
PULL alter
SAY 'Sie sind etwa' alter*365 'Tage alt.'
```

Speichern Sie dieses Programm unter dem Namen REXX: Alter.rexx und rufen Sie es mit dem folgenden Kommando auf:

```
RX alter
```

Dieses Programm beginnt mit einer Kommentarzeile, die die Aufgabe des Programms beschreibt. Alle ARexx-Programme beginnen mit einem Kommentar. Der Befehl SAY dient zur Anzeige einer Eingabeaufforderung im Konsolenfenster.

Der Befehl PULL liest die Eingabe des Benutzers - in diesem Fall also das Alter. PULL nimmt die Eingabe auf, wandelt sie gegebenenfalls in Großbuchstaben um und speichert sie in einer Variablen ab. Variablen sind Symbole, denen ein Wert zugewiesen werden kann. Es empfiehlt sich, beschreibende Variablennamen zu wählen. Hier wird der Name "alter" als Platzhalter für die vom Benutzer einzugebende Zahl verwendet.

Die letzte Zeile multipliziert die Variable "alter" mit 365 (Anzahl der Tage pro Jahr - Schaltjahre werden in diesem Beispiel ignoriert) und zeigt über den Befehl SAY das Ergebnis an. Beachten Sie, daß die Variable "alter" nicht als Zahl deklariert werden mußte, da ihr Wert zum Zeitpunkt der Verwendung im Ausdruck überprüft wird. (Hierbei handelt es sich um ein Beispiel für typenlose Daten.) Wenn Sie sehen möchten, was passiert, wenn sie für "alter" keine Zahl angeben, rufen Sie das Programm noch einmal auf und geben diesmal Buchstaben statt numerischer Zeichen für das Alter ein. Es erscheint eine Fehlermeldung, die die Zeilennummer und die Art des aufgetretenen Fehlers angibt.

2.3.3 Calc.rexx

In diesem Programm wird der Befehl DO vorgestellt, mit dem Programmanweisungen wiederholt ausgeführt werden können. Ferner veranschaulicht es auch den Exponentialoperator (**). Geben Sie das Programm über den Texteditor bzw. das Textverarbeitungsprogramm ein und speichern Sie es unter dem Namen REXX:Calc.rexx. Verwenden Sie das Kommando "RX calc" zum Aufrufen des Programms.

Programm 3. Calc.rexx

```
/*Berechnung von Quadraten und Kuben.*/
DO i = 1 to 10     /*Schleifenkopf – 10 Durchläufe*/
   SAY i i**2 i**3    /*Berechnungen und Ausgabe*/
   END                /*Schleifenende*/
SAY 'fertig.'
```

Der Befehl DO bewirkt, daß die zwischen den Befehlen DO und END stehenden Anweisungen wiederholt ausgeführt werden. Die Variable "i" ist die Indexvariable für die Programmschleife und erhöht sich bei jeder Wiederholung (Iteration) um 1. Die Zahl, die auf das Symbol TO folgt, ist der Endwert für den Befehl DO und könnte statt der Konstanten 10 auch eine Variable oder ein vollständiger Ausdruck sein.

Allgemein verwenden ARexx-Programme einzelne Leerzeichen zwischen verschiedenen Befehlsbestandteilen. In Programm 3 wurde allerdings auf Leerzeichen zwischen den Exponentialzeichen (**) und den Variablen und Konstanten (i und 2, i und 3) verzichtet.

Beachten Sie, daß die Anweisungen in der Schleife eingerückt sind. Für die Programmiersprache ist dies nicht erforderlich, es verbessert jedoch die Lesbarkeit des Programms, da Sie Anfang und Ende der Programmschleife auf einen Blick erkennen können.

2.3.4 Gerade.rexx

Der Befehl IF ermöglicht die Ausführung einer Anweisung unter einer bestimmten Bedingung. Im vorliegenden Beispiel werden die Zahlen von 1 bis 10 als gerade oder ungerade klassifiziert, indem sie durch zwei dividiert und der Rest anschließend überprüft wird. Der arithmetische Operator // berechnet den Rest (Modulo), der nach Ausführung der Division verbleibt.

Programm 4. Gerade.rexx

```
/*Gerade oder ungerade?*/
DO i = 1 to 10 /*Schleife beginnen - 10 Durchläufe*/
   IF i // 2 = 0 THEN typ = 'gerade'
     ELSE typ = 'ungerade'
   SAY i 'ist' typ
   END            /*Schleife beenden*/
```

Die IF-Zeile gibt an, daß im Falle eines Rests von 0 bei Division der Variable "i" durch 2 die Variable "typ" auf gerade gesetzt wird. Ist der Rest ungleich 0, überspringt das Programm den THEN-Zweig und führt stattdessen den ELSE-Zweig aus, d. h. die Variable "typ" wird auf ungerade gesetzt.

2.3.5 Quadrat.rexx

Dieses Beispiel stellt das Konzept einer Funktion vor, d. h. einer Gruppe von Anweisungen, die durch Angabe des Funktionsnamens in einem geeigneten Zusammenhang ausgeführt werden können. Mit Hilfe von Funktionen können Sie umfangreiche und komplexe Programme aus kleineren Modulen (Programmbausteinen) erstellen. Funktionen lassen auch die Verwendung desselben Programmtextes für ähnliche Operationen in verschiedenen Programmteilen zu.

Funktionen werden in einem Ausdruck als Name dargestellt, auf den eine öffnende Klammer folgt. (Zwischen Name und Klammer steht kein Leerzeichen.) Ein oder mehrere Ausdrücke, die als Argumente bezeichnet werden, können auf die Klammer folgen. Nach dem letzten Argument muß eine schließende Klammer folgen. Diese Argumente leiten Informationen zur Verarbeitung an die Funktion weiter.

Programm 5. Quadrat.rexx

```
/*Definieren und Aufrufen einer Funktion.*/
DO i = 1 to 5
   SAY i quadrat(i) /*Aufruf der Funktion "quadrat"*/
   END
EXIT
quadrat:            /*Funktionsname*/
   ARG x            /*Argument abrufen*/
   RETURN x**2 /*quadrieren und zurückgeben*/
```

Beginnend mit DO und endend auf END wird eine Schleife mit einer Indexvariablen "i" eingerichtet, die jeweils um 1 hochzählt. Die Schleife wird fünfmal durchlaufen. Die Schleife enthält einen Ausdruck, der die Funktion "quadrat" aufruft, wenn der Ausdruck ausgewertet wird. Das Resultat der Funktion wird mittels des Befehls SAY angezeigt.

Die Funktion "quadrat" ist durch die Befehle ARG und RETURN definiert und berechnet die Quadratwerte. ARG ruft den Wert aus der Argumentzeichenfolge "i" ab, RETURN leitet das Resultat der Funktion zurück zum aufrufenden Befehl SAY.

Sobald die Funktion von der Schleife aufgerufen wird, sucht das Programm nach dem Funktionsnamen "quadrat:", ruft anschließend das Argument "i" ab, führt die Rechenoperation aus und kehrt zum Anfang der DO/END-Schleife zurück. Der Befehl EXIT beendet das Programm nach dem letzten Scheifendurchlauf.

2.3.6 Results.rexx

Der Befehl TRACE aktiviert die Ablaufverfolgungsfunktion von ARexx.

Programm 6. Results.rexx

```
/*Darstellung einer "Ergebnis"-Ablaufverfolgung*/
TRACE results
sum = 0 ; sumq = 0;
DO i = 1 to 5
   sum = sum + 1
   sumq = sumq + i**2
   END
SAY 'sum=' sum 'sumq=' sumq
```

An der Konsole werden die Ausführung der Zeilen des Quellprogramms für jeden Durchlauf der DO/END-Schleife und anschließend die Endergebnisse des Ausdrucks angezeigt. Würde der Befehl TRACE entfernt, würde nur das Endergebnis angezeigt: sum = 15 sumq = 55.

2.3.7 Zensur.rexx

Dieses Programm berechnet die Abschlußzensur für einen bestimmten Schüler auf der Basis der erzielten Noten in vier Klausuren und einer Zensur für die mündliche Beteiligung. Die Durchschnittsnote aus Klausur 1 und Klausur 2 soll 30 % zur Abschlußnote beitragen, die Durchschnittsnote aus Klausur 3 und Klausur 4 soll 45 % ausmachen, und die mündliche Beteiligung wird mit 25 % veranschlagt. Beachten Sie, daß bei solchen Zahlenwerten der amerikanische Dezimalpunkt statt eines Kommas geschrieben werden muß.

Nachdem das Programm die Abschlußzensur angezeigt hat, fragt es den Benutzer, ob er eine weitere Note berechnen will. Die Antwort wird eingelesen ("PULL"). Lautet sie nicht Q (Quittieren = Programm verlassen), wird die Schleife fortgesetzt. Lautet die Antwort Q, wird die Schleife verlassen und der Programmablauf beendet.

Programm 7. Zensur.rexx

```
/*Zensurenprogramm*/
SAY "Hallo, ich berechne die Zensur für Sie."
response = 0
DO while response ~ = "Q"
   /*Schleife, solange Antwort nicht Q ist*/
   SAY "Bitte alle Einzelzensuren eingeben."
   SAY "Klausur 1:"
   PULL es1
   SAY "Klausur 2:"
   PULL es2
   SAY "Klausur 3:"
   PULL es3
   SAY "Klausur 4:"
   PULL es4
   SAY "Mündliche Beteiligung:"
   PULL p
   Note = (((es1 + es2)/2*.3) + ((es3 + es4)/2*.45) + (p*.25))
   SAY "Die Abschlußzensur ist" Note
   SAY "Wollen Sie weitermachen? (Mit Q beenden.)"
   PULL response
   END
EXIT
```

Kapitel 3

Elemente der Programmiersprache ARexx

In diesem Kapitel werden die Regeln und Konzepte der Programmiersprache ARexx behandelt. Außerdem wird erläutert, wie ARexx die in Programmen verwendeten Zeichen und Wörter interpretiert. Die hier behandelten Elemente umfassen:

- Token — das kleinste Element der ARexx-Sprache
- Klauseln — die kleinste ausführbare Einheit - vergleichbar einem Satz
- Ausdrücke — eine Gruppe ausgewerteter Tokens
- Die Kommandoschnittstelle — der Prozeß, über den ARexx-Programme mit ARexx-kompatiblen Anwendungen kommunizieren

Das Kapitel enthält auch eine Beschreibung der Ausführungsumgebung von ARexx. Dieser Abschnitt richtet sich an erfahrene Amiga-Benutzer und enthält technische Details zur Prozeßkommunikation.

3.1 Token

Token sind die kleinsten separaten Einheiten der Programmiersprache ARexx. Sie können aus einem oder mehreren Zeichen bestehen. Token können in fünf Kategorien unterteilt werden:

- Kommentare
- Symbole
- Zeichenfolgen
- Operatoren
- Sonderzeichen

3.1.1 Kommentare

Jede Gruppe von Zeichen, die mit der Sequenz /* beginnt und mit */ endet, bildet einen Kommentar. Jedes ARexx-Programm muß mit einem Kommentar beginnen. Zu jedem /* muß ein entsprechendes */ vorhanden sein. Beispiel:

```
/*Ein einfacher Kommentar in ARexx*/
```

Kommentare können überall im Programm plaziert und auch ineinander verschachtelt werden. Beispiel:

```
/*Ein /*verschachtelter*/ Kommentar*/
```

Es empfiehlt sich, Programme mit zahlreichen Kommentaren auszustatten, die Ihnen und anderen Benutzern Zweck und Funktion des jeweiligen Programms in Erinnerung rufen. Der Kommandointerpreter ignoriert Kommentare beim Lesen der Programmdateien - die Kommentare können also keine Verzögerung der Programmausführung auslösen.

3.1.2 Symbole

Unter einem Symbol versteht man eine beliebige Gruppe der Zeichen a-z, A-Z, 0-9 sowie Punkt (.), Ausrufezeichen (!), Fragezeichen (?) Dollarzeichen ($) und Unterstreichung (_). Wenn der Interpreter ein Programm durchliest, werden Symbole in Großbuchstaben übersetzt. Das Symbol MeinName entspricht also MEINNAME. Vier Symbolarten werden erkannt:

Feste Symbole	Eine Gruppe numerischer Zeichen, die mit einer Ziffer (0-9) oder einem Punkt (.) beginnt. Der Wert eines festen Symbols ist stets der Symbolname selbst, übersetzt in Großbuch-staben. 12345 ist ein Beispiel für ein festes Symbol.
Einfache Symbole	Eine Gruppe alphabetischer Zeichen, die mit einem Buchstaben (A-Z) beginnt, z. B. "MeinName".
Stammsymbole	Eine Gruppe alphanumerischer Zeichen, die auf einen Punkt endet, z. B. "A." und "Stamm9."
Zusammengesetzte Symbole	Eine Gruppe alphanumerischer Zeichen, die mehrere Punkte enthält, z. B. "A.1.Index".

Einfache, zusammengesetzte und Stammsymbole werden als Variablen bezeichnet. Ihnen kann während des Programmlaufs ein Wert zugewiesen werden. Solange eine Variable noch nicht mit einem Wert besetzt ist, ist sie nicht initialisiert. Für nicht initialisierte Variablen wird als Wert der Variablenname selbst (ggf. übersetzt in Großbuchstaben) verwendet.

Zusammengesetzte und Stammsymbole besitzen besondere Eigenschaften, die sie besonders für den Aufbau von Variablenfeldern und Listen prädestinieren. Stammsymbole ermöglichen die Initialisierung einer ganzen Klasse zusammengesetzter Symbole. Ein zusammengesetztes Symbol hat die Struktur Stamm.n_1.n_2 . . . n_k, wobei der Ausgangsname ein Stammsymbol ist und jeder Knoten, n_1. . . n_k, ein festes oder einfaches Symbol.

Wenn einem Stammsymbol ein Wert zugeordnet wird, ordnet es seinerseits diesen Wert allen vom Stammsymbol abgeleiteten, zusammengesetzten Symbolen zu. Der Wert eines zusammengesetzten Symbols hängt also von den zuvor auf dieses Symbol oder den dazugehörigen Stamm erfolgten Zuordnungen ab.

Wenn ein zusammengesetztes Symbol in einem Programm vorkommt, wird sein Name erweitert, indem jeder Knoten durch seinen aktuellen Wert ersetzt wird. Die Wertzeichenfolge kann aus beliebigen Zeichen bestehen (einschließlich eingefügter Leerzeichen) und wird nicht in Großbuchstaben umgewandelt. Resultat der Erweiterung ist ein neuer Name, der anstelle des zusammengesetzten Symbols verwendet wird. Hat z. B. J den Wert 3 und K den Wert 7, wird das zusammengesetzte Symbol A.J.K zu A.3.7 erweitert.

Zusammengesetzte Symbole können als eine Form zuordnungsabhängigen oder inhaltsadressierbaren ("assoziativen") Speichers betrachtet werden. Nehmen wir einmal an, Sie müssen eine Reihe von Namen und Telefonnummern speichern und abrufen. Die normale Vorgehensweise wäre nun, zwei Felder (NAME und NUMMER) einzurichten und jedem eine Indexnummer (1 bis n, mit n = Gesamtanzahl der Einträge) zu geben. Eine Nummer würde gesucht, indem man das Namensfeld durchliest, bis der gewünschte Name gefunden wird, z. B. NAME.12, dessen Telefonnummer NUMMER.12 dann im anderen Feld abgelesen werden kann. Arbeitet man mit zusammengesetzten Symbolen, könnte das Symbol NAME den abzurufenden Namen enthalten, und NUMMER.NAME würde zur dazugehörigen Telefonnummer erweitert, z. B. NUMMER.CBM.

Zusammengesetzte Symbole können auch als gewöhnliche indizierte Felder verwendet werden, die den zusätzlichen Vorteil bieten, daß nur eine Zuweisung (auf den Stamm) erforderlich ist, um die gesamte Feldgruppe zu initialisiereng Das Programm im folgenden Beispiel verwendet die Stämme "nummer." und "adr." zur Erstellung eines elektronischen Telefonverzeichnisses.

Programm 8. TelNr.rexx

```
/*Telefonverzeichnis zur Veranschaulichung
zusammengesetzter Variablen.*/
IF ARG() ~ = 1 THEN DO
SAY "VERWENDUNG: rx TelNr Name"
   EXIT 5
   END
/*Zur Anzeige von Telefonnummern/Adressen Fenster
öffnen.*/
CALL OPEN out, "con:0/0/640/60/ARexx Phonebook"
IF ~ result THEN DO
   SAY "Öffnen gescheitert ... Sorry"
   EXIT 10
   END
/*Nummerndefinitionen*/
nummer. = '(nicht gefunden)'
nummer.wsh = '(555) 001-0001'
adr. = '(nicht gefunden)'
nummer.CBM = '(555) 002-0002'
adr.CBM = '1200 Wilson Dr., West Chester, PA, 19380'

/*(Eigentliche Arbeit schon getan)*/
ARG name   /*Der Name*/
CALL WRITELN out,name || "TelNr   ist" nummer.name
CALL WRITELN out,name || "Adresse ist" adr.name
CALL WRITELN out, "Mit Eingabetaste Programm
verlassen."
CALL READLN out
EXIT
```

Zum Aufrufen des Programms rufen Sie ein Shell-Fenster auf und geben folgendes ein:

```
RX TelNr cbm
```

Es erscheint ein Fenster mit dem CBM zugeordneten Namen und der Adresse.

3.1.3 Zeichenfolgen

Eine Zeichenfolge ist eine beliebige Gruppe von Zeichen, an deren Anfang und Ende einfache (') oder doppelte (") Anführungszeichen als Begrenzungszeichen stehen. Am Ende der Zeichenfolge muß allerdings immer das gleiche Begrenzungszeichen stehen wie am Anfang. Wenn das Begrenzungszeichen selbst Bestandteil der Zeichenfolge sein soll, muß es zweimal nacheinander angegeben werden ('' oder ""). Beispiel:

```
"Hallo Kumpel,"
'haste mal ''ne Mark?'
```

Der Wert einer Zeichenfolge ist die Zeichenfolge selbst. Die Länge einer Zeichenfolge wird durch die Anzahl der Zeichen bestimmt. Wenn die Zeichenfolge keine Zeichen enthält, wird sie als leere Zeichenfolge (Nullzeichenfolge) bezeichnet.

Wenn auf eine Zeichenfolge unmittelbar ein X oder B folgt, handelt es sich um eine hexadezimale bzw. binäre Zeichenfolge. Sie muß sich aus hexadezimalen (0-9, A-F) bzw. binären Ziffern (0,1) zusammensetzen. Beispiel:

```
'4A 3B C0'X
'00110111'B
```

Leerzeichen sind an Bytegrenzen aus Gründen der Übersichtlichkeit zulässig. Hexadezimale und binäre Zeichenfolgen eignen sich besonders für nicht im ASCII-Code definierte Zeichen und für maschinenspezifische Informationen, z. B. Adressen in einem Programm. Sie werden umgehend in die gepackte (komprimierte), maschineninterne Form umgewandelt.

3.1.4 Operatoren

Operatoren sind eine Kombination aus folgenden Zeichen: ~ + – * / = > < & | ^. Darauf wird in diesem Abschnitt näher eingegangen. Man unterscheidet vier Arten von Operatoren:

- Arithmetische Operatoren erfordern einen oder zwei numerische Operanden und ergeben ein numerisches Resultat.

- Verkettungsoperatoren verbinden zwei Zeichenfolgen zu einer Zeichenfolge.
- Vergleichsoperatoren erfordern Operanden und ergeben ein boolesches Resultat (0 oder 1).
- Logische Operatoren erfordern einen oder zwei boolesche Operanden und ergeben ein boolesches Resultat.

Jedem Operator ist ein Prioritätswert zugeordnet, der die Reihenfolge bestimmt, in der die Operationen in einem Ausdruck ausgeführt werden. Operatoren mit höherer Priorität (8) werden vor solchen mit niedriger Priorität (1) ausgeführt.

3.1.4.1 Arithmetische Operatoren

Eine wichtige Klasse der Operatoren sind solche, die Zahlen darstellen. Zahlen bestehen aus den Zeichen 0-9, Punkt (.), Pluszeichen (+), Minuszeichen (–), und Leerzeichen. Zur Darstellung der Exponentialschreibweise kann auf eine Zahl ein "e" oder "E" und eine ganze Zahl (mit Vorzeichen) folgen.

Zur Angabe von Zahlen können Zeichenfolgen und Symbole verwendet werden. Da die Sprache typenlos ist, müssen Variablen nicht vor der Verwendung in arithmetischen Operationen als numerisch deklariert werden. Statt dessen wird jede Wertzeichenfolge bei der Verarbeitung daraufhin überprüft, ob sie eine Zahl darstellt. Die folgenden Beispiele stellen alle gültige Zahlen dar:

```
33
"  12.3  "
0.321e12
' + 15. '
```

Voran- und nachgestellte Leerzeichen sind zulässig. Leerzeichen können zwischen einem Plus- oder Minuszeichen (+ oder -) und der darauffolgenden Zahl stehen, aber nicht innerhalb der Zahl selbst.

Die für arithmetische Berechnungen zu verwendende Basisgenauigkeit kann während des Ablaufs eines Programms verändert werden. Die Anzahl geltender Stellen, die in arithmetischen Operationen verwendet werden, richtet sich nach der Vorgabe für Numeric Digits (Numerische Ziffern) und kann über den Befehl NUMERIC verändert werden. Näheres dazu siehe Kapitel 4.2.

Die Anzahl der für ein Resultat zulässigen Dezimalstellen hängt von der Operation und den Dezimalstellen der beteiligten Operanden ab. ARexx verwendet ggf. abschließende Nullen zur Anzeige der Genauigkeit des Resultats. Wenn die Gesamtzahl der zur Wiedergabe eines Wertes erforderlichen Ziffern größer ist als die Voreinstellung für Numeric Digits, wird die Zahl in Exponentialschreibweise formatiert. Dafür gibt es folgende Möglichkeiten:

- Wissenschaftliche Schreibweise — der Exponent wird so angepaßt, daß links vom Dezimalkomma nur eine Ziffer steht.
- Technische Schreibweise — die Zahl wird skaliert, so daß der Exponent ein Vielfaches von 3 ist und die Ziffern links vom Dezimalkomma im Bereich von 1 bis 999 liegen.

Tabelle 3-1 zeigt eine Liste arithmetischer Operatoren.

Tabelle 3-1. Arithmetische Operatoren

Operator	Priorität	Beispiel	Resultat
+ (Vorzeichen)	8	+'3.12'	3.12
– (negatives Vorzeichen)	8	–"3.12"	–3.12
**** (Potenzierung)**	7	0.5 ** 3	0.125
*** (Multiplikation)**	6	1.5*1.50	2.250
/ (Division)	6	6 / 3	2
% (Ganzzahldivision)	6	–8 % 3	–2
// (Rest)	6	5.1//0.2	0.1
+ (Addition)	5	3.1+4.05	7.15
– (Subtraktion)	5	5.55 – 1	4.55

3.1.4.2 Verkettungsoperatoren

ARexx definiert zwei Verkettungsoperatoren. Der erste, der durch die Operatorzeichenfolge | | (zwei senkrechte Striche) gekennzeichnet ist, verbindet zwei Zeichenfolgen zu einer Zeichenfolge, ohne daß zwischen den Ausgangszeichenfolgen ein Leerzeichen eingefügt wird. Diese Art der Verkettung kann auch implizit angegeben werden. Wenn ein Symbol und eine Zeichenfolge ohne jedes Leerzeichen eingegeben werden, verhält sich ARexx so, als

wäre der Operator | | angegeben worden. Die zweite Verkettungsoperation ist durch den Leerzeichenoperator gekennzeichnet und fügt die zwei Operandenzeichenfolgen zusammen, wobei zwischen beide ein Leerzeichen gesetzt wird.

Alle Verknüpfungsoperationen haben die Priorität 4. Tabelle 3-2 faßt die verschiedenen Operationen zusammen.

Tabelle 3-2. Verkettungsoperatoren

Operator	Operation	Beispiel	Resultat
||	Verkettung	'Is was,'||'Doc?'	Is was,Doc?
Leerzeichen	Verkettung mit Leerzeichen	'Guten' 'Tag'	Guten Tag
Keiner	Implizite Verkettung	eins'zwei'drei	EINSzweiDREI

3.1.4.3 Vergleichsoperatoren

ARexx unterstützt drei Arten von Vergleichen:

- Exakte Vergleiche — der Vergleich erfolgt zeichenweise
- Zeichenfolgenvergleich — führende Leerzeichen werden ignoriert, an die kürzere Zeichenfolge werden Leerzeichen angefügt
- Numerische Vergleiche — die Operanden werden auf der Basis der Voreinstellung für Numeric Digits in ein internes numerisches Format umgewandelt; anschließend wird ein arithmetischer Vergleich ausgeführt

Vergleiche ergeben stets einen booleschen Wert. Die Ziffern 0 und 1 dienen zur Darstellung der booleschen Werte "false" und "true" ("falsch" und "wahr"). Wenn ein boolescher Operand erwartet wird, führt die Verwendung eines anderen Wertes als 0 oder 1 zu einem Fehler. Als boolescher Wert wird auch jede Zahl akzeptiert, die 0 oder 1 äquivalent ist, z. B. 0.000 oder 0.1E1.

Mit Ausnahme der Operatoren für exakte Gleichheit (==) und exakte Ungleichheit (~==) bestimmen alle Vergleichsoperatoren dynamisch, ob ein Zeichenfolgenvergleich oder ein numerischer Vergleich auszuführen ist. Ein numerischer Vergleich erfolgt, wenn

beide Operanden gültige Zahlen sind. Andernfalls werden die Operanden als Zeichenfolgen verglichen.

Alle Vergleiche haben die Priorität 3. Tabelle 3-3 zeigt eine Liste der zulässigen Vergleichsoperatoren.

Tabelle 3-3. Vergleichsoperatoren

Operator	Operation	Modus
==	Exakte Gleichheit	Exakt
~==	Exakte Ungleichheit	Exakt
=	Gleichheit	Zeichenfolge/Numerisch
~=	Ungleichheit	Zeichenfolge/Numerisch
>	Größer als	Zeichenfolge/Numerisch
>= oder ~<	Größer als oder gleich	Zeichenfolge/Numerisch
<	Kleiner als	Zeichenfolge/Numerisch
<= oder ~>	Kleiner als oder gleich	Zeichenfolge/Numerisch

3.1.4.4 Logische (Boolesche) Operatoren

ARexx definiert die vier logischen Operationen NICHT, UND, ODER und Exklusiv-ODER. Alle diese Operationen benötigen boolesche Operanden und ergeben ein boolesches Resultat. Der Versuch, mit anderen als booleschen Operanden eine logische Operation auszuführen, führt zu einem Fehler. Tabelle 3-4 zeigt eine Liste der zulässigen logischen Operatoren.

Tabelle 3-4. Logische Operatoren

Operator	Priorität	Operation
~	8	Logisches NICHT (Umkehrung)
&	2	Logisches UND
|	1	Logisches ODER
^ oder &&	1	Logisches Exklusiv-ODER

3.1.5 Sonderzeichen

Einige Interpunktionszeichen besitzen innerhalb von ARexx eine spezielle Bedeutung, wie Tabelle 3-5 veranschaulicht.

Tabelle 3-5. Sonderzeichen

Sonderzeichen	Definition
(:) Doppelpunkt	Ein Doppelpunkt, dem ein Symbol-Token (ein alphanumerisches Zeichen oder . ! ? $) vorangestellt ist, definiert eine Sprungmarke in einem Programm.
( ) Klammern	Klammern werden in Ausdrücken dazu verwendet, Operatoren und Operanden in Unterausdrücken zusammenzufassen und so die normalen Prioritätswerte der betreffenden Operatoren aufzuheben. Eine öffnende Klammer dient auch zur Kennzeichnung eines Funktionsaufrufs innerhalb eines Ausdrucks. Ein Symbol oder eine Zeichenfolge, worauf unmittel-bar eine öffnende Klammer folgt, definiert einen Funktionsnamen. Innerhalb einer Anweisung muß die Anzahl öffnender und schließender Klammern jeweils gleich sein.
(;) Semikolon	Das Semikolon dient als Endezeichen einer Anweisung. Wenn Sie mehrere kurze Anweisungen haben, die in eine Zeile passen, werden diese durch Semikolons voneinander getrennt.
(,) Komma	Das Komma dient als Fortsetzungszeichen am Zeilenende für Anweisungen, die in mehrere Zeilen aufgeteilt wurden, und als Trennzeichen zwischen Argumentausdrücken innerhalb eines Funktionsaufrufs.

3.2 Klauseln

Eine Klausel wird aus Token-Gruppen gebildet und stellt die kleinste als Anweisung ausführbare Spracheinheit dar.

Beim Lesen des Programms teilt der Sprachinterpreter das Programm in Klauselgruppen auf. Diese Gruppen mit einer oder mehreren Klauseln werden dann weiter in Token aufgeteilt, und

jede Klausel wird dann unter einem bestimmten Typ klassifiziert. Dabei können unscheinbare syntaktische Unterschiede den semantischen Inhalt einer Anweisung vollständig verändern. Beispiel:

```
SAY 'Hallo, Willi'
```

ist eine Befehlsklausel und zeigt "Hallo, Willi" auf der Konsole an, aber:

```
''SAY 'Hallo, Willi'
```

ist eine Kommandoklausel, und gibt "SAY Hallo, Willi" als Kommando an ein externes Programm aus. Die führende Nullzeichenfolge (' ') wirkt sich also auf die Klassifizierung aus: aus einer Befehlsklausel wird eine Kommandoklausel.

Das Zeilenende gilt in der Regel implizit als Ende einer Klausel. Eine Klausel kann jedoch auch in der folgenden Zeile fortgesetzt werden. Dazu muß die erste Zeile mit einem Komma abgeschlossen werden. Das Komma wird vom Programm ignoriert, und die nächste Zeile wird als Fortsetzung der Klausel interpretiert. Für solche Fortsetzungen gibt es kein festgelegtes Maximum (abgesehen davon, daß die Größe des Kommandopuffers nicht überschritten werden darf).

Zeichenfolgen- und Kommentar-Token werden automatisch fortgesetzt, wenn die Zeile vor Erreichen des abschließenden Begrenzungszeichens beendet ist. Das Zeichen für Neue Zeile (das mit der Eingabetaste generiert wird) wird nicht als Bestandteil des Tokens betrachtet.

3.2.1 Null-Klauseln

Zeilen, die lediglich aus Leerzeichen oder Kommentaren bestehen, werden als Null-Klauseln bezeichnet. Sie können an jeder beliebigen Stelle in einem Programm vorkommen. Für die Ausführung des Programms sind sie ohne Bedeutung. Sie verbessern jedoch die Übersichtlichkeit des Programms und erhöhen die Zeilenzahl.

3.2.2 Sprungmarkenklauseln

Ein Symbol, auf das ein Doppelpunkt (:) folgt, wird als Sprungmarkenklausel bezeichnet. Eine Sprungmarke dient als Markierungszeichen in einem Programm. Die Ausführung einer Sprungmarke hat jedoch keine Aktion zur Folge. Der Doppelpunkt wird als implizites Beendigungszeichen einer Klausel betrachtet, d. h. jede Sprungmarke bildet eine separate Klausel. Sprungmarkenklauseln können an jeder beliebigen Stelle im Programm vorkommen. Beispiel:

```
start:    /*Ausführung beginnen*/
syntax:   /*Fehlerbehandlung*/
```

3.2.3 Zuweisungsklauseln

Zuweisungsklauseln werden durch ein Variablensymbol gekennzeichnet, auf das der Operator = folgt. (In diesem Kontext wird die normale Definition des Operators = - Prüfung auf Gleichheit - aufgehoben.) Die Token rechts vom Gleichheitszeichen werden als Ausdruck ausgewertet, und das Ergebnis wird der Variablen zugeordnet. Beispiel:

```
when = 'Jetzt ist es soweit'
answ = 3.14 * fact(5)
```

Das Gleichheitszeichen (=) weist den Wert 'Jetzt ist es soweit' der Variablen 'when' und das Ergebnis der Rechenoperation 3.14 * fact(5) der Variablen 'answ' zu.

3.2.4 Befehlsklauseln

Befehlsklauseln beginnen mit dem Namen des Befehls und weisen ARexx zur Ausführung einer Aktion an. Befehlsnamen werden in Kapitel 4 beschrieben. Beispiel:

```
DROP a b c
SAY 'bitte'
IF j > 5 THEN LEAVE;
```

3.2.5 Kommandoklauseln

Kommandoklauseln sind alle ARexx-Ausdrücke, die nicht unter die bisher beschriebenen Klauselkategorien fallen. Der Ausdruck wird ausgewertet, und das Resultat wird als Kommando an einen externen Host (s. u., "Gastgeberprogramm") ausgegeben. Beispiel:

```
'delete' 'Datei'  /*Ein AmigaDOS-Befehl*/
'jump' current+10 /*Ein Editorbefehl*/
```

Der Befehl "delete" wird nicht als ARexx-Kommando erkannt und daher an einen externen Host - in diesem Fall AmigaDOS - gesendet. Der Befehl "jump" im zweiten Beispiel wird normalerweise von einem externen Texteditor verstanden.

3.3 Ausdrücke

Unter Ausdrücken versteht man eine Gruppe ausgewerteter Tokens. Die meisten Anweisungen enthalten zumindest einen Ausdruck. Ausdrücke setzen sich aus folgenden Komponenten zusammen:

- Zeichenfolgen — Der Wert einer Zeichenfolge ist die Zeichenfolge selbst.
- Symbole — Der Wert eines festen Symbols ist das Symbol selbst, umgewandelt in Großbuchstaben. Symbole können als Variablen verwendet werden und einen zugewiesenen Wert haben.
- Operatoren — Operatoren besitzen einen Prioritätswert, der festlegt, an welcher Stelle innerhalb der Gesamtreihenfolge die einzelnen Operatoren ausgeführt werden.
- Klammern — Runde Klammern können verwendet werden, um die Reihenfolge der Auswertung in einem Ausdruck zu verändern oder um Funktionsaufrufe zu kennzeichnen. Ein Symbol oder eine Zeichenfolge, auf das/die unmittelbar eine öffnende Klammer folgt, definiert den Namen der Funktion, die Tokens zwischen öffnender und schließender Klammer bilden die Argumentliste der Funktion. So setzt sich z. B. der Ausdruck:

```
J 'Fakultät ist' fact(J)
```

aus folgenden Komponenten zusammen:

- ein Symbol — J
- ein Leeroperator
- eine Zeichenfolge — Fakultät ist
- ein weiteres Leerzeichen
- ein Symbol — fact
- eine öffnende Klammer
- ein Symbol — J
- eine schließende Klammer

In diesem Beispiel ist FACT der Funktionsname und (J) die dazugehörige Argumentenliste, die in diesem Fall nur den einzelnen Ausdruck J umfaßt.

Bevor die Auswertung eines Ausdrucks stattfinden kann, benötigt ARexx einen Wert für jedes Symbol in dem Ausdruck. Bei festen Symbolen ist dieser Wert der Symbolname selbst, variable Symbole müssen dagegen in der aktuellen Symboltabelle nachgesehen werden. Im obigen Beispiel würde der Ausdruck unter der Voraussetzung, daß dem Symbol J der Wert 3 zugewiesen wird, wie folgt aussehen:

```
3 'Fakultät ist' FACT(3)
```

Zur Vermeidung von Unklarheiten bezüglich der Werte, die den Symbolen während des Auflösungsprozesses zugewiesen werden, garantiert ARexx die strikte Einhaltung der Auflösungsreihenfolge von links nach rechts. Die Symbolauflösung wird ungeachtet der Priorität des Operators und der Anordnung der Klammern fortgesetzt. Wird ein Funktionsaufruf festgestellt, wird die Auflösung während der Auswertung der Funktion ausgesetzt. Beachten Sie bitte, daß ein und dasselbe Symbol innerhalb eines Ausdrucks mehr als einen Wert annehmen kann.

Nehmen wir an, das obige Beispiel würde wie folgt umgestellt:

```
FACT(J) 'ist' J 'Fakultät'
```

Würde nun das zweite J ebenfalls in den Wert 3 aufgelöst? Allgemein können Funktionsaufrufe Nebeneffekte haben, z. B. die

Veränderung der Variablenwerte. Der Aufruf FACT im umgestellten Beispiel könnte also eine Veränderung des Werts von J zur Folge haben, so daß beim zweiten Vorkommen von J schon der neue Wert wirksam wird.

Nach Auflösung aller Symbolwerte wird der Ausdruck ausgehend von den Prioritätswerten der Operatoren und der Anordnung der Klammerausdrücke ausgewertet. Bei der Auswertung von Operatoren mit gleichem Prioritätswert garantiert ARexx keine bestimmte Reihenfolge. Ferner verwendet ARexx bei der Auswertung boolescher Operationen keine "Abkürzungen". In dem Ausdruck:

```
(1 = 2) & (FACT(3) = 6)
```

wird z. B. die Funktion FACT aufgerufen, obwohl der erste Term der UND- (&-) Operation 0 ist. An diesem Beispiel wird deutlich, daß ARexx mit dem Lesen des Programms von links nach rechts fortfährt, obwohl die erste Klammer im Beispiel logisch falsch ist und einen Wert von 0 ergibt.

3.4 Die Kommandoschnittstelle

Die ARexx-Kommandoschnittstelle ist ein allgemein zugänglicher Message-Port. Zu ARexx kompatible Anwendungen müssen über diesen Message-Port verfügen. ARexx-Programme geben Kommandos aus, indem sie die Kommandozeichenfolge in ein Message-Paket stellen und dieses Paket an den Message-Port des Hosts (Gastprogramms) senden. Das Programm setzt die Operation aus, solange der Host die Kommandos verarbeitet. Wenn das Nachrichtenpaket zurückgemeldet wird, wird die Operation wieder aufgenommen.

3.4.1 Host-Adresse

ARexx verwaltet zwei implizite Host (Gastprogramm) - Adressen - einen aktuellen und einen vorhergegangenen Wert - als Bestandteil der Speicherumgebung des Programms. Diese Werte können über den Befehl ADDRESS jederzeit geändert werden (oder über das Synonym SHELL). Die aktuelle Host-Adresse kann mit Hilfe der

integrierten Funktion ADDRESS() abgefragt werden. Die Standardzeichenfolge für die Host-Adresse ist REXX, diese kann jedoch bei Aufruf eines Programms neu gesetzt werden. Die meisten Host-Anwendungen liefern den Namen ihres allgemein zugänglichen Ports mit, wenn sie ein Makro-Programm aufrufen, so daß das Makro automatisch Kommandos zurück an den Host ausgeben kann.

Eine spezielle Host-Adresse wird erkannt. Die Zeichenfolge COMMAND gibt an, daß das Makro direkt an AmigaDOS ausgegeben werden sollte. Bei allen anderen Host-Adressen wird davon ausgegangen, daß sie sich auf einen öffentlichen Message-Port beziehen. Der Versuch, ein Kommando an einen nicht existenten Message-Port zu senden, führt zu einem Syntaxfehler ("Host-Umgebung nicht gefunden").

Programm 9 zeigt die Interaktion zwischen ARexx und dem AmigaDOS-Editor, ED. Das Programm stellt fest, ob ED läuft, bestimmt den Namen des Message-Ports und richtet einige Stammvariablen ein.

Programm 9. ED-Status.rexx

```
/*Gibt den Status von ED aus. ED muß laufen, bevor
dieses Programm gestartet wird. ED-Ports heißen 'Ed',
'Ed_1', 'Ed_2' usw.*/
DEFAULT_ED = "Ed"  /*Groß-/Kleinschreibung ist hier
relevant*/
/*Prozedur für den Fall, daß ED nicht oder nur als
Zweitaufruf (oder noch höher) läuft.*/
DO WHILE ~ SHOW('p',DEFAULT_ED)      /*Port suchen*/
  SAY "Nicht gefunden: Port mit Namen" DEFAULT_ED
  SAY "Verfügbare Ports:"
  SAY SHOW('P') '0a'X
  SAY "Anderen Port-Namen eingeben oder mit QUIT
  Programm verlassen"
  /* Der Benutzer soll den Port wählen, wenn das
  Programm ihn nicht findet */
  DEFAULT_ED = READLN(stdin)
  IF STRIP(UPPER(DEFAULT_ED)) = 'QUIT' then exit 10
  /*Benutzer kann Programm hierdurch verlassen*/
  END
SAY "Benutzt wird ED-Port" DEFAULT_ED
/*Nun, da der Port gefunden ist, soll ARexx ihn
adressieren.*/
ADDRESS VALUE DEFAULT_ED
/*Einige nützliche Stammvariablen einrichten*/
STEM.0 = 15              /*Anzahl ED-ARexx-Variablen*/
STEM.1 = 'LEFT'                  /*Linker Rand (SL)*/
STEM.2 = 'RIGHT'                /*Rechter Rand (SR)*/
STEM.3 = 'TABSTOP'              /*Tabulatoren (ST)*/
STEM.4 = 'LMAX'          /*Maximal anzeigbare Zeilen*/
STEM.5 = 'WIDTH'    /*Breite der Anzeige in Zeichen*/
STEM.6 = 'X'     /*Phys. X-Pos. am Bildschirm ab 1*/
STEM.7 = 'Y'     /*Phys. Y-Pos. am Bildschirm ab 1*/
STEM.8 = 'BASE'                       /*Fensterbasis*/
       /*Basis ist 0, es sei denn, die Anzeige wird*/
                          /*nach rechts verschoben*/
STEM.9 = 'EXTEND'          /*Randwert erweitern (EX)*/
STEM.10 = 'FORCECASE'      /*Groß-/Kleinschreibung*/
STEM.11 = 'LINE'           /*Aktuelle Zeilenzahl*/
STEM.12 = 'FILENAME'              /*Datei wird ediert*/
STEM.13 = 'CURRENT'       /*Text der aktuellen Zeile*/
STEM.14 = 'LASTCMD'  /*Letztes erweitertes Kommando*/
STEM.15 = 'SEARCH'         /*Letzte Suchzeichenfolge*/
/*ED soll Werte in Stammvariable 'STEM.' stellen.*/
'RV' '/STEM/'    /*RV ist ein ED-Kommando zum Senden
von Infos von ED an ARexx*/
```

```
/*STEM.1 ist LEFT, und STEM.LEFT hat nun einen Wert
von ED. So können diese Daten ausgedruckt werden:*/

DO i = 1 to STEM.0
ED_VAR = STEM.i
SAY STEM.i "=" STEM.ED_VAR /*ED-Variablen und Werte
drucken*/
END
```

3.4.2 Erstellen eines Makros

ARexx kann zum Schreiben von Programmen für jede Host-Anwendung eingesetzt werden, die über eine kompatible Kommandoschnittstelle verfügt. Manche Anwendungsprogramme sind auf der Basis einer integrierten Makro-Sprache geschrieben und können zahlreiche vordefinierte Makro-Kommandos anbieten.

Überprüfen Sie das Anwendungsprogramm auf "Shortcut"-Kommandos (Tastenbefehle). Manche Programme bieten leistungsstarke Funktionen, die speziell für den Aufruf von Makro-Programmen aus implementiert wurden.

Die Interpretation der empfangenen Kommandos hängt ausschließlich von der jeweiligen Host-Anwendung ab. Im einfachsten Fall entsprechen die Kommandozeichenfolgen exakt den Kommandos, die vom Benutzer eingegeben werden können. So werden z. B. die Positionssteuerkommandos (Auf/Ab) eines Texteditors wahrscheinlich gleich interpretiert. Andere Kommandos sind aber möglicherweise nur gültig, wenn sie von einem Makro aus gegeben werden. Ein Kommando zur Simulation einer Menüoperation würde wohl kaum von einem Benutzer eingegeben. In Beispiel 10 wird das ARexx-Programm von ED zur Vertauschung zweier Zeichen aufgerufen.

Programm 10. Transpose.rexx

```
/*Gegebene Zeichenfolge '123', wenn Cursor auf 3 ist,
wandelt das Makro die Zeichenkette in '213' um.*/

HOST = ADDRESS()    /*Von welchem ED kam der Aufruf?*/
ADDRESS VALUE HOST
/*. . . mit Editor kommunizieren.*/
'rv' '/CURR/'  /*ED soll Info in Stamm CURR stellen*/

/*Wir brauchen zwei Angaben:*/
currpos = CURR.X      /*Cursorposition in der Zeile*/
currlin = CURR.CURRENT
/*Inhalt der aktuellen Zeile*/

IF (currpos >2)    /*Mindestens 3. Schreibstellen*/
   THEN currpos = currpos - 1
   ELSE DO    /*Fehler melden und Programm verlassen*/
   'sm /Cursor muß mind. auf Pos. 3 stehen/'
   EXIT 10
END

/*Zeichen CURRPOS und CURRPOS-1 vertauschen und
aktuelle Zeile durch neue Zeile ersetzen.*/
DROP CURR. /* STEM-Variable CURR wird nicht mehr
gebraucht; Speicherplatz einsparen */

'd'                /*ED soll aktuelle Zeile löschen*/
currlin = swapch (currpos,currlin)  /*2 Zeichen
tauschen*/
'i /'||currlin||'/'      /*Geänderte Zeile einfügen*/
DO i = 1 to currpos    /*Cursor zurück an alte
Stelle*/
   'cr'      /*ED-Kommando 'Cursor nach rechts'*/
   END
EXIT      /*Alles erledigt*/
/*Funktion zum Austauschen zweier Zeichen*/
swapch: procedure
PARSE ARG cpos,clin
   chl = substr (clin, cpos, 1)    /*Zeichen holen*/
   clin = delstr (clin, cpos, 1)  /*Löschen aus
    Zeichenfolge*/
   clin = insert (chl,clin,cpos-2,1)  /*Einfügen, um
     Transposition zu komplettieren*/
RETURN clin    /*Geänderte Zeichenfolge zurückgeben*/
```

Zum Aufrufen dieses Beispiels von ED aus drücken Sie bitte ESC und geben folgendes ein:

```
RX "Rexx:transpose.rexx"
```

Hier muß die volle Pfadangabe und die Namenserweiterung angegeben werden.

Diese Zeichenfolge kann natürlich auch einer Funktionstaste zugeordnet werden.

3.4.3 Rückgabecodes

Nach Abschluß der Kommandoverarbeitung meldet der Host den Kommandostatus in Form eines Rückgabecodes. Die bei den einzelnen Kommandos möglichen Rückgabecodes lesen Sie bitte in der Dokumentation zur Host-Anwendung nach. Diese Codes zeigen an, ob die vom Kommando ausgeführte Operation erfolgreich war.

Dieser Rückgabecode wird in die ARexx-Spezialvariable RC gestellt, damit er vom Makro geprüft werden kann. Der Wert Null zeigt an, daß das Kommando erfolgreich ausgeführt wurde. Eine positive ganze Zahl weist auf eine Fehlerbedingung hin. Je höher die Zahl, desto schwerer der Fehler. Aus dem Rückgabecode kann das Makro-Programm schließen, ob das Kommando ausgeführt wurde bzw. ob im Falle des Scheiterns eine Aktion erforderlich ist.

3.4.4 Kommando-Shells

ARexx ist zwar so aufgebaut, daß mit Programmen, die seine spezifische Kommandoschnittstelle unterstützen, am effektivsten gearbeitet werden kann, es kann aber grundsätzlich mit jedem Kommando-Shell-Programm eingesetzt werden, das standardmäßige E/A-Mechanismen zum Abrufen seines Datenstroms unterstützt. Eine Möglichkeit, ARexx einzusetzen, besteht darin, eine Befehlsdatei in der RAM-Disk zu erstellen und diese direkt an die Shell weiterzuleiten. In Programm 11 wird eine neue Shell zur Ausführung eines standardmäßigen EXECUTE-Skripts geöffnet:

Programm 11. Shell.rexx

```
/*Neue Shell starten*/
ADDRESS command
conwindow = "CON:0/0/640/100/NeueShell/Close"

/*Befehlsdatei erstellen*/
CALL OPEN out,"ram:temp",write
CALL WRITELN out, 'echo "Dies ist ein Test"'
CALL CLOSE out

/*Neues Shell-Fenster öffnen*/
'newshell' conwindow "ram:temp"
EXIT
```

3.5 Die Ausführungsumgebung

Hinweis Die folgenden Informationen richten sich an erfahrene Amiga-Benutzer. Sie setzen ein Grundlagenwissen über das Amiga-Betriebssystem und die Kenntnis der Amiga ROM Kernel Manuals voraus.

Der ARexx-Interpreter RexxMast bietet eine einheitliche Ausführungsumgebung, indem er jedes Programm als separaten Prozeß im Multitasking-Betriebssystem des Amiga ausführt. Damit verfügen Sie über eine flexible Schnittstelle zwischen einem externen Host-Programm und RexxMast. Das Host-Programm kann seine Operationen entweder gleichzeitig ausführen oder warten, bis das interpretierte ARexx-Programm abgeschlossen ist. Jedes ARexx-Programm besitzt eine externe und eine interne Umgebung.

3.5.1 Externe Umgebung

Zur externen Umgebung eines Programms gehören seine Prozeßstruktur, die Ein- und Ausgabeströme sowie das aktuelle Verzeichnis. Bei der Erstellung jedes ARexx-Prozesses übernimmt dieser die Ein- und Ausgabeströme von seinem Klienten, also dem externen Programm, das das ARexx-Programm aufrief. Wurde z. B. ein ARexx-Programm von einer Shell aus gestartet, übernimmt das ARexx-Programm die Ein- und Ausgabeströme sowie das aktuelle

Verzeichnis dieser Shell. Die Suche nach einer Programm- oder Datendatei geht dann vom aktuellen Verzeichnis aus. Für externe Funktionen gilt ein Maximum von 15 Argumenten.

3.5.2 Interne Umgebung

Die interne Umgebung eines ARexx-Programms besteht aus einer statischen globalen Struktur und einer oder mehreren Speicherumgebungen. Die globalen Datenwerte sind zum Zeitpunkt des Programmaufrufs feste Werte (statisch). Zu diesen Werten gehören der Quelltext des Programms, statische Datenzeichenfolgen und Argumentzeichenfolgen. Sobald das Programm läuft, können diese Werte nicht mehr geändert werden.

ARexx-Programme, die als Kommandos aufgerufen werden, besitzen in der Regel nur eine Argumentzeichenfolge, auch wenn die Möglichkeit zur Aufspaltung des Kommandos (Tokenisierung) die Möglichkeit zur Verwendung mehrerer Argumentzeichenfolgen bietet. Ein als interne Funktion aufgerufenes Programm kann eine beliebige Anzahl Argumente besitzen. Diese Argumente bleiben für die Dauer des Programmablaufs bestehen.

Die Speicherumgebung umfaßt die Symboltabelle, die für Variablenwerte benutzt wird, numerische Optionen, Ablaufverfolgungsoptionen und Host-Adreßzeichenfolgen. Während der Programmausführung kann es nur eine globale Umgebung, aber mehrere Speicherumgebungen geben. Bei jedem Aufruf einer internen Funktion wird eine neue Speicherumgebung aktiviert und initialisiert. Die Ausgangswerte für die meisten Felder werden aus der vorherigen Umgebung übernommen. Werte können jedoch später geändert werden, ohne daß sich dies auf die Umgebung der Aufrufebene auswirkt. Die neue Umgebung bleibt so lange bestehen, bis die Funktion die Steuerung zurückgibt.

Zu jeder Speicherumgebung gehört eine Symboltabelle. Sie dient zum Speichern der Wertzeichenfolgen, die den Variablen zugewiesen werden. Diese Symboltabelle ist in Form eines aus zwei Ebenen bestehenden, binären Baums aufgebaut. Die Primärebene speichert Einträge für einfache Symbole und Stammsymbole, die Sekundärebene wird für zusammengesetzte Symbole verwendet. Alle zusammengesetzten Symbole, die den gleichen Stamm besitzen,

werden in einem Baum gespeichert, wobei der Eintrag für den Stamm die Wurzel des Baumes bildet.

Symbole werden erst nach einer erfolgten Zuweisung in die Tabelle aufgenommen. Einmal auf der Primärebene erstellte Einträge werden niemals gelöscht, selbst wenn die Initialisierung des Symbols später rückgängig gemacht wird. Sekundäre Bäume werden freigegeben, wenn eine Zuweisung an den zu diesem Baum gehörenden Stamm erfolgt.

3.5.3 Betriebsmittelverwaltung

ARexx bietet eine vollständige Verwaltung aller dynamisch zugeordneten Betriebsmittel (Resource-Tracking), die zur Ausführung eines Programms verwendet werden. Zu diesen Ressourcen zählen Speicherplatz, DOS-Dateien und verwandte Strukturen sowie die Message-Port-Struktur. Dieses Verwaltungssystem ist so ausgelegt, daß bei einem Programmabbruch zu einem beliebigen Zeitpunkt keine blockierten Betriebsmittel zurückgelassen werden.

Durch direkte Aufrufe des Amiga-Betriebssystems von einem ARexx-Programm aus ist es möglich, das System der Betriebsmittelverwaltung des Interpreters zu übergehen. In diesem Fall ist es Aufgabe des Programmierers, alle Betriebsmittel, die sich außerhalb des ARexx-Betriebsmittelverwaltungssystems befinden, zu kontrollieren und zurückzugeben. ARexx bietet eine spezielle Interrupt-Einrichtung, so daß ein Programm nach einem Ausführungsfehler die Steuerung behalten, die erforderliche Bereinigung ausführen und ordnungsgemäß beendet werden kann.

Kapitel 4
Befehle

Eine Befehlsklausel beginnt mit dem Namen eines bestimmten Befehls und weist ARexx an, eine bestimmte Aktion auszuführen. Dieses Kapitel enthält eine Liste der in ARexx verfügbaren Befehle.

Auf jedes Befehlsschlüsselwort können ein oder mehrere Unterschlüsselwörter, Ausdrücke oder andere befehlsspezifische Informationen folgen. Befehlsschlüsselwörter und Unterschlüsselwörter werden nur in diesem spezifischen Kontext als solche erkannt. Daher ist es möglich, die gleichen Schlüsselwörter in verschiedenen Kontexten als Variablen oder Funktionsnamen zu verwenden. Auf ein Befehlsschlüsselwort darf keiner der Operatoren Doppelpunkt (:) oder Gleichheitszeichen (=) folgen.

4.1 Syntax

Die Syntax jeder Anweisung steht jeweils rechts vom Schlüsselwort in der Überschrift. Tabelle 4-1 zeigt die in den Syntaxangaben geltenden Konventionen:

Tabelle 4-1. Syntaxkonventionen

Konvention	Definition
SCHLÜSSELWORT	Alle Schlüsselwörter und Unterschlüsselwörter sind in Großbuchstaben geschrieben
ausdruck	Erforderliche Argumente sind in Kleinbuchstaben geschrieben
| (senkrechter Strich)	Alternative Auswahlmöglichkeiten sind durch einen senkrechten Strich voneinander (bzw. von der Standardauswahl) getrennt
{} (geschweifte Klammern)	Erforderliche Alternativen stehen in geschweiften Klammern
[] (eckige Klammern)	Wahlweise verwendbare Befehlsteile stehen in eckigen Klammern

Der Befehl CALL hat z. B. folgendes Format:

```
CALL {Symbol | Zeichenfolge} [Ausdruck]
[,Ausdruck,...]
```

Als Argument müssen Sie ein Symbol oder eine Zeichenfolge angeben. Der senkrechte Strich kennzeichnet Alternativen, zwischen denen gewählt werden kann. Die geschweiften Klammern bedeuten, daß ein Argument in jedem Fall erforderlich ist. Die Angabe eines Ausdrucks ist nicht zwingend erforderlich, was Sie an den eckigen Klammern erkennen können.

Am Ende jeder Befehlsbeschreibung ist jeweils ein Beispiel angegeben. Erläuterungen oder Bewertungen der Beispiele finden Sie in Form von ARexx-Kommentarzeilen (/*. . . */).

4.2 Befehle in alphabetischer Reihenfolge

Dieser Abschnitt enthält eine Liste integrierter ARexx-Befehle in alphabetischer Reihenfolge. Die Syntax jedes Befehls sehen Sie jeweils rechts vom Befehlsschlüsselwort.

ADDRESS ADDRESS [[Symbol | Zeichenfolge] | [WERT][Ausdr.]]

Dieser Befehl gibt eine Host-Adresse für vom Interpreter ausgegebene Kommandos an. Unter einer Host-Adresse versteht man den Namen des Message-Ports einer Anwendung, an den ARexx-Kommandos gesendet werden. ARexx verwaltet zwei Host-Adressen: einen aktuellen und einen vorhergehenden Wert. Immer wenn Sie eine neue Host-Adresse angeben, wird diese zum aktuellen Wert und die bisher aktuelle Adresse zur vorhergehenden. Der bis dahin vorhergehenden Wert wird gelöscht. Die Host-Adressen sind Bestandteil der Speicherumgebung eines Programms und überdauern auch interne Funktionsaufrufe. Die aktuelle Adresse kann mit der integrierten Funktion ADDRESS() aufgerufen werden.

Das Schlüselwort ADDRESS - alleine angegeben - schaltet vom aktuellen auf den vorhergehenden Host um, bei wiederholter Ausführung würde also zwischen den zwei Host-Adressen hin- und hergeschaltet.

ADDRESS {Zeichenfolge | Symbol} macht die/das angegebene Zeichenfolge/Symbol zur neuen Host-Adresse. Der Wert der Zeichenfolge bzw. des Symbols ist also das Token selbst. Bei Namen von Message-Ports wird zwischen Groß- und Kleinschreibung unterschieden. Ein Programmkommando an den Message-Port MeinPort müßte also wie folgt aussehen:

```
ADDRESS 'MeinPort'
```

Werden die einfachen Anführungszeichen weggelassen, sucht ARexx nach dem Message-Port MEINPORT und gibt eine Fehlermeldung aus. Die aktuelle Host-Adresse wird zur vorhergehenden. Ein auf die Zeichenfolge bzw. das Symbol folgender Ausdruck wird

ausgewertet, das Ergebnis wird an den angegebenen Host gesendet. Aktuelle und vorhergehende Adresse bleiben unverändert. Auf diese Weise kann ein einzelnes Kommando an einen externen Host ausgegeben werden, ohne daß die Host-Adressen davon berührt werden. Der Rückgabewert des Kommandos wird behandelt wie der einer Kommandoklausel.

Bei Angabe von ADDRESS [VALUE] Ausdruck verwendet ARexx das Resultat des Ausdrucks als neue Host-Adresse (d. h. die bisher aktuelle Adresse wird zur vorhergehenden). Das Schlüsselwort VALUE kann weggelassen werden, wenn das erste Token des Ausdrucks weder ein Symbol noch eine Zeichenfolge ist. Beispiel:

```
ADDRESS   /*Wechsel zwischen aktueller und
   vorhergehender Adresse.*/
ADDRESS edit    *Die neue Host-Adresse lautet EDIT.*/
ADDRESS edit 'top'          /*An den Anfang gehen.*/
ADDRESS VALUE edit in /*Eine neue Host-Adresse
   berechnen.*/
```

ARG ARG [Schablone] [,Schablone ...]

ARG ist eine Kurzform des Befehls PARSE UPPER ARG. Damit können eine oder mehrere dem Programm zur Verfügung stehende Argumentzeichenfolgen abgerufen und den Variablen in der Schablone zugeordnet werden. Die Anzahl der verfügbaren Argumentzeichenfolgen hängt davon ab, ob das Programm als Kommando oder als Funktion aufgerufen wurde. Kommandoaufrufe verfügen in der Regel nur über eine Argumentzeichenfolge, Funktionen können dagegen bis zu 15 aufweisen. Die Argumentzeichenfolgen werden durch den Befehl ARG nicht verändert. ARG gibt Großbuchstaben zurück. Beispiel:

```
ARG erstes,zweites  /*Argumente abrufen*/
```

Struktur und Verarbeitung von Schablonen werden im Abschnitt zum Befehl PARSE auf Seite 4-15 kurz beschrieben.

BREAK BREAK

Der Befehl BREAK dient dazu, den Bereich eines DO-Befehls oder einer INTERPRETierten Zeichenfolge zu verlassen. Er ist nur in einem solchen Kontext gültig. Wird er innerhalb einer DO-Anweisung verwendet, wird die innerste DO-Anweisung, die den Befehl BREAK enthält, verlassen. Im Gegensatz dazu wird mit dem verwandten Befehl LEAVE nur eine iterative (d. h. sich wiederholende) DO-Anweisung verlassen. Beispiel:

```
DO              /*Block beginnen*/
   IF a>3 THEN BREAK /*Fertig?*/
   a = a + 1
   y.a = name
   END           /*Block beenden*/
```

CALL CALL {Symbol | Zeichenfolge} [Ausdruck] [,Ausdruck, ...]

Der Befehl CALL dient zum Aufrufen einer internen oder externen Funktion. Der Funktionsname wird durch das Symbol- oder Zeichenfolge-Token angegeben. Alle darauf folgenden Ausdrücke werden ausgewertet und werden zu Argumenten der aufgerufenen Funktion. Der von der Funktion zurückgegebene Wert wird der Sondervariablen RESULT zugeordnet. Es gilt nicht als Fehler, wenn keine Ergebniszeichenfolge zurückgegeben wird. In diesem Fall wird die Variable RESULT gelöscht, d. h. ihre Initialisierung wird mittels der Anweisung DROP rückgängig gemacht.

Die Verbindung zur Funktion wird zum Zeitpunkt des Aufrufs dynamisch hergestellt. Bei der Suche nach einer aufgerufenen Funktion hält ARexx eine bestimmte Reihenfolge ein. Beispiel:

```
CALL CENTER name,laenge+4,'+'
```

CENTER ist die aufgerufene Funktion. Die Ausdrücke werden ausgewertet und als Argumente an CENTER weitergeleitet.

DO DO [[Var=Ausdr] | [Ausdr] [TO Ausdr] [BY Ausdr]] [FOR Ausdr] [FOREVER] [WHILE Ausdr | UNTIL Ausdr]

Mit der Anweisung DO wird eine Befehlsgruppe eingeleitet, die als zusammengehöriger Block ausgeführt wird. Der Bereich des Befehls DO umfaßt alle Anweisungen bis zu und einschließlich eines möglichen Befehls END.

Wenn auf den Befehl DO kein Unterschlüsselwort folgt, wird der Block einmal ausgeführt. Durch Angabe von Unterschlüsselwörtern kann die Ausführung eines Blocks wiederholt werden, bis eine Endebedingung eintritt. Eine iterative (wiederholte) DO-Anweisung wird manchmal auch als Schleife (engl. loop) bezeichnet, da ARexx sozusagen mit einem "Looping" an eine bereits durchlaufene Stelle im Programm zurückkehrt, um einen Befehl nochmals auszuführen. Der Befehl DO setzt sich aus folgenden Komponenten zusammen:

- Ein Initialisierungsausdruck im Format "Variable=Ausdruck" definiert die Indexvariable der Schleife. Der Ausdruck wird ausgewertet, wenn der Bereich des Befehls DO erstmals aktiviert wird. Das Ergebnis wird der Indexvariablen zugeordnet. Bei folgenden Wiederholungen (Iterationen) wird ein Ausdruck im Format "Variable = Variable + Inkrement" ausgewertet, wobei das Inkrement (die Zunahme) das Ergebnis des Ausdrucks BY ist. Wird ein Initialisierungsausdruck angegeben, muß er allen anderen Unterschlüsselwörtern vorausgehen.
- Der auf ein Symbol BY folgende Ausdruck definiert das Inkrement, das bei jeder folgenden Iteration zur Indexvariablen addiert wird. Der Ausdruck muß ein numerisches Resultat ergeben, das positiv oder negativ sein kann und keine ganze Zahl sein muß. Das Standardinkrement ist 1.
- Das Resultat des Ausdrucks nach TO legt die obere (oder untere) Grenze für die Indexvariable fest. Bei jeder Iteration wird die Indexvariable mit dem Ergebnis von TO verglichen. Ist das Inkrement (Resultat von TO) positiv und die Variable größer als der Grenzwert, bricht der Befehl DO ab und übergibt die Steuerung an die auf den Befehl END folgende Anweisung. Die Schleife wird auch beendet, wenn das Inkrement negativ ist und die Indexvariable unter dem TO-Grenzwert liegt.

- Der Ausdruck nach FOR muß bei der Auswertung eine positive ganze Zahl ergeben. Er legt die maximale Anzahl der auszuführenden Iterationen fest. Die Schleife wird beendet, sobald dieser Grenzwert erreicht ist, unabhängig vom Wert der Indexvariablen.
- Die Initialisierungsausdrücke BY, TO und FOR werden nur bei der ersten Aktivierung des Befehls ausgewertet; damit bleiben das Inkrement und die Grenzwerte während der gesamten Ausführung unverändert. Ein Grenzwert ist nicht unbedingt erforderlich. So kann z. B. mit dem Befehl "DO i=1" ein Zählvorgang unbegrenzt lange weiterlaufen.
- Das Schlüsselwort FOREVER kann verwendet werden, wenn ein iterativer DO-Befehl gebraucht wird, aber keine Indexvariable notwendig ist. Die Schleife kann durch einen in der Schleife enthaltenen Befehl LEAVE oder BREAK beendet werden.
- Der Ausdruck nach WHILE wird am Anfang jeder Iteration ausgewertet und muß einen booleschen Wert ergeben. Die Iteration wird bei Resultat 1 (oder "wahr") fortgesetzt. Andernfalls wird der Schleifendurchlauf beendet.
- Der Ausdruck nach UNTIL wird am Ende jeder Iteration ausgewertet und muß einen booleschen Wert ergeben. Der Befehl wird mit der nächsten Iteration fortgesetzt, wenn das Resultat 0 (oder "falsch") lautet, andernfalls wird er abgebrochen. (WHILE und UNTIL schließen sich gegenseitig aus.)

Programm 12. Iteration.rexx

```
/*Beispiele für DO*/
LIMIT = 20; nummer = 1
DO i=1 to LIMIT for 10 WHILE nummer < 20
   nummer = i * nummer
   SAY "Iteration" i "nummer=" nummer
   END
nummer = nummer/3.345; i = 0
DO nummer for LIMIT/5
   i = i + 1
   SAY "Iteration" i "nummer=" nummer
   END
```

Die Ausgabe wird hier mit erläuternden Kommentarzeilen gezeigt, die am Bildschirm natürlich nicht erscheinen würden.

```
Iteration 1 Nummer = 1    /* 1 * 1 = 1 */
Iteration 2 Nummer = 2    /* 2 * 1 = 2 */
Iteration 3 Nummer = 6    /* 3 * 2 = 6 */
Iteration 4 Nummer = 24      /*4 * 6 = 24 */
Iteration 1 Nummer = 7.17488789  /* 24/3.345 =
7.17488789 */
Iteration 2 Nummer = 7.17488789  /* Nummer bleibt
gleich */
Iteration 3 Nummer = 7.17488789  /* limit/5 = 20/5 =
4 */
Iteration 4 Nummer = 7.17488789  /* Operation wird 4
mal wiederholt */
```

Hinweis Ist auch ein FOR-Grenzwert vorhanden, wird der ursprüngliche Ausdruck dennoch ausgewertet, muß aber keine positive ganze Zahl ergeben.

DROP DROP Variable [Variable ...]

Die angegebenen Variablensymbole werden auf den Stand vor ihrer Initialisierung rückgesetzt. Auf diesem Stand entspricht der Wert der Variablen dem Variablennamen selbst. Es liegt kein Fehler vor, wenn eine Variable, die nicht mehr initialisiert ist, mit dem Befehl DROP gelöscht wird. Wird ein Stammsymbol mit DROP gelöscht, werden damit auch die Werte aller vom Stammsymbol abgeleiteten zusammengesetzten Symbole gelöscht. Beispiel:

```
a = 123    /*a einen Wert zuordnen */
DROP a b   /*Werte von A und B löschen (mit DROP)*/
SAY a b    /*Resultat ist: A B.*/
```

ECHO ECHO [Ausdruck]

Der Befehl ECHO ist gleichbedeutend mit dem Befehl SAY. Er zeigt das Ergebnis des Ausdrucks am Konsolenbildschirm an. Beispiel:

```
ECHO "Was Du nicht sagst!"
```

ELSE ELSE [;] [bedingte Anweisung]

Der Befehl ELSE leitet den alternativen bedingten Zweig einer IF-Anweisung ein. Er ist nur innerhalb des Bereichs eines IF-Befehls gültig und muß auf die bedingte Anweisung des THEN-Zweiges folgen. Wurde der THEN-Zweig nicht ausgeführt, wird die auf die Klausel ELSE folgende Anweisung verarbeitet.

ELSE-Klauseln beziehen sich stets auf die unmittelbar davor befindliche IF-Anweisung. In manchen Fällen ist es notwendig, "Pseudo" ELSE-Klauseln für die inneren IF-Bereiche mehrerer verschachtelter IF-Anweisungen anzugeben, damit auch die äußeren IF-Anweisungen über alternative Verzweigungen verfügen können. Es genügt nicht, die ELSE-Anweisung mit einem Semikolon oder einer Null-Klausel abzuschließen. Stattdessen kann der Befehl NOP (No-operation) verwendet werden. Beispiel:

```
IF i > 2 THEN SAY 'Tatsächlich?'
   ELSE SAY 'Habe ich mir gedacht'
```

END END [Variable]

Der Befehl END schließt den Bereich eines DO- oder SELECT-Befehls ab. Wenn das wahlfreie Variablensymbol angegeben wird, wird es mit der Indexvariablen der DO-Anweisung verglichen (die iterativ sein muß). Stimmen die Symbole nicht überein, kommt es zu einem Fehler. Beispiel:

```
DO i=1 to 5    /*Indexvariable ist i*/
   SAY i
   END i       /*"i"-Schleife beenden*/
```

EXIT EXIT [Ausdruck]

Der Befehl EXIT beendet die Ausführung eines Programms. Er ist an jeder beliebigen Stelle in einem Programm gültig. Der ausgewertete Ausdruck wird als Ergebnis der Funktion oder des Kommandos an die Aufrufebene zurückgegeben.

Die Verarbeitung des EXIT-Ergebnisses hängt davon ab, ob das aufrufende Programm eine Ergebniszeichenfolge angefordert hat

und ob der aktuelle Aufruf aus einem Kommando- oder Funktionsaufruf hervorging:

- Wurde eine Ergebniszeichenfolge angefordert, wird das Ergebnis des Ausdrucks in einen eigens reservierten Speicherblock kopiert. Ein Verweiszeiger auf diesen Block wird als Sekundärergebnis dieses Aufrufs zurückgegeben.
- Forderte dagegen die Aufrufebene keine Ergebniszeichenfolge an und wurde das Programm als Kommando aufgerufen, wird versucht, das Ergebnis des Ausdrucks in eine ganze Zahl umzuwandeln. Dieser Wert wird dann als Primärergebnis zurückgegeben. Das Sekundärergebnis ist in diesem Falle 0. Auf diese Weise können die EXIT-Informationen von der Aufrufebene als Rückgabewert interpretiert werden.

Beispiel:

```
EXIT            /*Kein Ergebnis erforderlich*/
EXIT 10         /*Eine Fehlerrückmeldung*/
```

IF IF Ausdruck [THEN] [;] [bedingte Anweisung]

Der Befehl IF wird in Verbindung mit den Befehlen THEN und ELSE zur bedingten Ausführung einer Anweisung verwendet. Das Ergebnis des Ausdrucks muß ein boolescher Wert sein. Ist das Ergebnis 1 ("Wahr"), wird die auf das Symbol THEN folgende Anweisung ausgeführt. Andernfalls geht die Steuerung an die nächste Anweisung über. Das Schlüsselwort THEN muß nicht unmittelbar auf den IF-Ausdruck folgen, sondern kann auch als separate Klausel vorliegen.

Der Befehl wird als "IF Ausdruck; THEN; Anweisung." analysiert. Der Ausdruck, der auf die IF-Anweisung folgt, stellt die Testbedingung dar, die entscheidet, ob im folgenden die Klausel THEN oder ELSE ausgeführt wird. Auf das Symbol THEN kann eine beliebige gültige Anweisung folgen. Insbesondere ermöglicht ein Block "DO . . . END;" die bedingte Ausführung mehrerer Anweisungen. Beispiel:

```
IF result < 0 THEN exit         /*Alles erledigt?*/
```

INTERPRET INTERPRET Ausdruck

Das Kommando INTERPRET behandelt den Ausdruck wie einen Block von Quellenanweisungen. Der Ausdruck wird ausgewertet, und das Ergebnis wird wie eine oder mehrere Programmanweisungen ausgeführt. Die Anweisungen werden als Block betrachtet, so als ob sie von "DO . . . END" umschlossen wären. Jede beliebige Anweisung kann in dem INTERPRETierten Quelltext enthalten sein, einschließlich der Befehle DO und SELECT. Der Befehl BREAK kann zur Beendigung der Verarbeitung INTERPRETierter Anweisungen eingesetzt werden.

Ein INTERPRET-Befehl aktiviert bei seiner Ausführung einen Steuerbereich, der als Begrenzung für die Befehle LEAVE und ITERATE dient. Diese Befehle können nur in DO-Schleifen verwendet werden, die innerhalb von INTERPRET definiert wurden. Es ist zwar kein Fehler, wenn Sprungmarkenklauseln innerhalb der interpretierten Zeichenfolge stehen. Zur Übertragung der Steuerung wird jedoch nur unter denjenigen Sprungmarken gesucht, die im Originalprogramm definiert wurden.

Der Befehl INTERPRET kann zur dynamischen Gestaltung und Ausführung von Programmen verwendet werden. Programmfragmente können als Argumente an Funktionen weitergegeben werden, die dann die Fragmente INTERPRETieren. Beispiel:

```
bef = 'SAY'              /*Ein Befehl*/
INTERPRET bef hallo      /*. . . "SAY HALLO"*/
```

ITERATE ITERATE [Variable]

Der Befehl ITERATE beendet die aktuelle Iteration (den Schleifendurchlauf) eines DO-Befehls und startet die nächste Iteration. Effektiv bedeutet dies, daß die Steuerung an die END-Anweisung und anschließend (abhängig vom Ergebnis des Ausdrucks UNTIL) wieder zurück an die DO-Anweisung übertragen wird. Der Befehl wirkt sich normalerweise auf den innersten iterativen DO-Bereich aus. Wenn sich der Befehl ITERATE nicht innerhalb eines iterativen DO-Befehls befindet, kommt es zu einem Fehler.

Falls mehrere verschachtelte Bereiche existieren, bestimmt das wahlfreie Variablensymbol, welcher DO-Bereich verlassen werden soll. Die Variable wird als Literal gelesen und muß mit der Indexvariablen eines gegenwärtig aktiven DO-Befehls übereinstimmen. Wenn kein entsprechender DO-Befehl gefunden wird, kommt es zu einem Fehler. Beispiel:

```
DO i=1 to 5
   IF i = 3 THEN ITERATE i
   SAY i
   END
```

LEAVE LEAVE [Variable]

LEAVE führt zum sofortigen Verlassen des iterativen DO-Bereichs, der diesen Befehl enthält. Befindet sich der Befehl LEAVE nicht innerhalb eines iterativen DO-Befehls, kommt es zu einem Fehler. Falls mehrere verschachtelte Bereiche existieren, bestimmt das wahlfreie Variablensymbol, welcher DO-Bereich verlassen werden soll. Die Variable wird als Literal gelesen und muß mit der Indexvariablen eines gegenwärtig aktiven DO-Befehls übereinstimmen. Wenn kein entsprechender DO-Befehl gefunden wird, kommt es zu einem Fehler. Beispiel:

```
DO i = 1 to limit
   IF i > 5 THEN LEAVE  /*Maximale Iterationen*/
   END
```

NOP NOP

Der Befehl NOP (keine Operation, engl. no operation) dient zur Anbindung von ELSE-Klauseln an verschachtelte IF-Anweisungen. Beispiel:

```
IF i = j THEN              /*Erstes (äußeres) IF*/
   IF j = k THEN a = 0     /*Inneres IF*/
      ELSE NOP             /*Anbindung ans innere IF*/
   ELSE a = a + 1          /*Anbindung ans äußere IF*/
```

NUMERIC NUMERIC {DIGITS | FUZZ} Ausdruck NUMERIC FORM {SCIENTIFIC | ENGINEERING}

- Der Befehl NUMERIC dient zur Einstellung von Optionen, die sich auf die numerische Genauigkeit und das Format beziehen. Die numerischen Optionen bleiben bei Aufruf einer internen Funktion erhalten.
- Die Ausdrucksoption DIGITS legt die Anzahl geltender Stellen für die Genauigkeit bei arithmetischen Berechnungen fest. Der Ausdruck muß bei der Auswertung eine positive ganze Zahl ergeben.
- Die Ausdrucksoption FUZZ legt die Anzahl geltender Stellen fest, die bei numerischen Vergleichsoperationen ignoriert werden sollen. Hierbei muß es sich um eine positive ganze Zahl handeln, die kleiner ist als die derzeitig aktuelle Option DIGITS.
- Die Option FORM SCIENTIFIC legt zur Exponentialdarstellung von Zahlen die wissenschaftliche Schreibweise fest. Der Exponent wird angepaßt, so daß die Mantisse (bei Zahlen ungleich Null) im Bereich von 1 bis 9 liegt. Dies ist das Standardformat.
- Die Option FORM ENGINEERING wählt zur Exponentialdarstellung von Zahlen die technische Schreibweise. Dabei wird eine Zahl normalisiert, so daß ihr Exponent ein Vielfaches von drei ist und die Mantisse (falls ungleich Null) im Bereich von 1 bis 999 liegt.

```
NUMERIC DIGITS 12    /*Auf 12 Ziffern genau*/
NUMERIC FORM SCIENTIFIC /*Resultat in
wissenschaftlicher Schreibweise*/
```

OPTIONS

OPTIONS [FAILAT Ausdruck]
OPTIONS [PROMPT Ausdruck]
OPTIONS [RESULTS]
OPTIONS [CACHE]

Der Befehl OPTIONS dient zum Setzen verschiedener interner Standardwerte. Der Ausdruck nach FAILAT setzt den Grenzwert, bei dessen Erreichen oder Überschreiten Rückgabewerte von Kommandos als Fehler angesehen werden. Die Auswertung des Ausdrucks muß eine ganze Zahl ergeben. Der Ausdruck nach PROMPT legt eine Zeichenfolge fest, die bei dem Befehl PULL (oder PARSE PULL) als Eingabeaufforderung verwendet wird. Das Schlüsselwort RESULTS gibt an, daß der Interpreter eine Ergebniszeichenfolge anfordern soll, wenn er Kommandos an einen externen Host ausgibt.

Die über diesen Befehl gesteuerten internen Optionen bleiben auch über Funktionsaufrufe hinweg erhalten, d. h. ein Befehl OPTIONS kann innerhalb einer internen Funktion gegeben werden, ohne dadurch die Umgebung der Aufrufebene zu beeinflussen. Wird kein Schlüsselwort zum Befehl OPTIONS angegeben, nehmen alle genannten Optionen wieder ihre Standardwerte an. Der Befehl OPTIONS akzeptiert auch das Schlüsselwort NO, um eine ausgewählte Option auf ihren Standardwert rückzusetzen. Dies erleichtert es beispielsweise, das Attribut RESULTS für ein einzelnes Kommando rückzusetzen, ohne zusätzlich auch die Optionen FAILAT und PROMPT rücksetzen zu müssen.

OPTIONS akzeptiert auch das Schlüsselwort CACHE, das zur Aktivierung/Inaktivierung eines internen Cache zur Zwischenspeicherung von Anweisungen dient. Normalerweise ist der Cache aktiviert. Beispiel:

```
OPTIONS FAILAT 10
OPTIONS PROMPT "Ja, Chef?"
OPTIONS RESULTS
```

OTHERWISE OTHERWISE [;] [bedingte Anweisung]

Dieser Befehl ist nur im Bereich eines SELECT-Befehls gültig und muß am Schluß aller "WHEN . . . THEN"-Anweisungen folgen. War keine der vorherigen WHEN-Klauseln zutreffend, wird die auf den Befehl OTHERWISE folgende Anweisung ausgeführt. In einem SELECT-Bereich ist ein OTHERWISE nicht unbedingt erforderlich. Allerdings kommt es zu einem Fehler, wenn die Klausel OTHERWISE weggelassen wird und keiner der WHEN-Befehle zutrifft. Beispiel:

```
SELECT
   WHEN i=1 THEN say 'eins'
   WHEN i=2 THEN say 'zwei'
   OTHERWISE SAY 'andere'
   END
```

PARSE PARSE [UPPER] Eingabequelle [Schablone] [,Schablone ...]

Der Befehl PARSE bietet einen Mechanismus zum Extrahieren einer oder mehrerer Teilzeichenfolgen aus einer Zeichenfolge und zum Zuweisen dieser Extrakte an Variablen. Die Eingabezeichenfolge kann aus verschiedenen Quellen stammen, z. B. aus Argumentzeichenfolgen, aus einem Ausdruck oder von der Konsole.

Die Syntaxanalyse (engl. Parsing) wird über eine Schablone gesteuert, die sich aus Symbolen, Zeichenfolgen, Operatoren und Klammern zusammensetzen kann. Die Schablone gibt sowohl die Variablen an, denen Werte zugewiesen werden sollen, als auch die Art der Begrenzung der Wertzeichenfolgen. Während des Analysevorgangs wird die Zeichenfolge in Unterzeichenfolgen aufgeteilt, die dann den Variablensymbolen in der Schablone zugeordnet werden. Der Prozeß dauert an, bis allen Variablen in der Schablone ein Wert zugeordnet wurde. Wenn die Eingabezeichenfolge vorher "aufgebraucht" ist, erhalten die verbleibenden Variablen Nullwerte.

Wenn auf eine Variable in der Schablone unmittelbar eine weitere Variable folgt, wird die Wertzeichenkette durch Aufspalten der Eingabezeichenfolge in Wörter (getrennt durch Leerzeichen) festgelegt. Führende und abschließende Leerzeichen sind nicht zulässig. Jedes Wort wird einer Variablen in der Schablone

zugewiesen. Normalerweise erhält dabei die letzte Variable den nicht tokenisierten Rest der Eingabezeichenfolge, da auf sie kein Symbol folgt. Ein Platzhaltersymbol, ein Punkt (.), führt dazu, daß die Variable mit dem Punkt endet, sobald ein Leerzeichen im Eingabestrom vorkommt. Platzhalter verhalten sich wie Variablen, allerdings wird ihnen niemals ein Wert zugewiesen.

Die Schablone kann weggelassen werden, wenn der Befehl nur zur Auswertung einer Eingabezeichenfolge vorgesehen ist. Schablonen werden in Kapitel 7.1 beschrieben.

Das Ziel der Analyseoperation ist es, für jedes Variablensymbol in der Schablone einen Bezug zu einer aktuellen und einer nächsten Position herzustellen. Die Unterzeichenfolge zwischen diesen Positionen wird dann der Variablen als Wert zugeordnet:

Im folgenden finden Sie eine Beschreibung der verschiedenen Optionen zu diesem Befehl:

- Das wahlfreie Schlüsselwort UPPER kann mit jeder Eingabequelle verwendet werden und bedeutet, daß die Eingabe-zeichenfolge vor der Analyse in Großbuchstaben umgewandelt werden soll. UPPER muß das erste Token sein, das auf PARSE folgt.
- Die Quellen für die Eingabezeichenfolgen werden durch die im folgenden erläuterten Schlüsselwortsymbole festgelegt. Bei Angabe mehrerer Schablonen empfängt jede Schablone eine neue Eingabezeichenfolge, auch wenn bei manchen Quellenoptionen die neue Zeichenfolge mit der vorherigen identisch ist. Die Zeichenfolge der Eingabequelle wird vor der Syntaxanalyse kopiert, d. h. die Originalzeichenfolgen werden durch den Analy-seprozeß nicht verändert.
- Die Eingabeoption ARG ruft die Argumentzeichenfolgen ab, die beim Aufruf des Programms abgerufen wurden. Kommando-aufrufe weisen in der Regel nur eine einzige Argument-zeichenfolge auf, Funktionen dagegen bis zu 15.
- Die Eingabezeichenfolge EXTERNAL wird aus dem STDERR-Strom gelesen (siehe Kapitel 6.1.1), um zu vermeiden, daß PUSH- oder QUEUE-Daten verändert werden. Bei Angabe mehrerer Schablonen liest jede Schablone eine neue Zeichenfolge. Diese Quellenoption entspricht der Option PULL.

- Die Eingabeoption NUMERIC stellt die aktuellen numerischen Optionen in eine Zeichenfolge, und zwar in der Reihenfolge DIGITS, FUZZ und FORM, jeweils getrennt durch ein einzelnes Leerzeichen.
- Die Eingabeoption PULL liest eine Eingabeoption von der Eingabekonsole. Bei Angabe mehrerer Schablonen liest jede Schablone eine neue Zeichenfolge.
- Die Eingabeoption SOURCE ruft die Quellenzeichenfolge für das Programm ab. Die Zeichenfolge wird wie folgt formatiert:

  ```
  {COMMAND|FUNCTION} {0|1} AUFGERUFEN AUFGELÖST ERW HOST
  ```

  Die einzelnen Komponenten haben folgende Bedeutung:

  - {COMMAND | FUNCTION} gibt an, ob das Programm als Kommando oder als Funktion aufgerufen wurde.
  - {0 | 1} ist ein boolesches Kennzeichen und gibt an, ob eine Ergebniszeichenfolge von der Aufrufebene angefordert wurde.
  - AUFGERUFEN steht für den Namen, unter dem dieses Programm aufgerufen wurde.
  - AUFGELÖST bezeichnet den endgültigen Namen des Programms (nach der Auflösung).
  - ERW steht für die bei Suchoperationen zu verwendende Dateinamenserweiterung (Standardwert ist "REXX").
  - HOST gibt die ursprüngliche Host-Adresse für Kommandos an.

Die Option SOURCE gibt nun den vollständigen Pfadnamen der ARexx-Programmdatei zurück. Zuvor wurde lediglich ein relativer Namen angegeben, was zur Lokalisierung der Quelldatei des Programms nicht ausreichte.

Die Eingabezeichenfolge "VALUE Ausdruck WITH" ist Ergebnis des angegebenen Ausdrucks. Das Schlüsselwort WITH ist erforderlich, um den Ausdruck von der Schablone zu trennen. Das Resultat des Ausdrucks kann durch die Verwendung mehrerer Schablonen mehrmals analysiert werden, der Ausdruck selbst wird dagegen nicht neu ausgewertet.

Die Eingabeoption "VAR Variable" verwendet den Wert der angegebenen Variable als Eingabezeichenfolge. Bei Verwendung mehrerer Schablonen verwendet jede Schablone den aktuellen Wert der Variablen. Dieser Wert kann sich ändern, wenn die Variable zu den Zuweisungszielen in einer der Schablonen gehört.

Die Eingabeoption VERSION der aktuellen Konfiguration des ARexx-Interpreters wird im folgenden Format angegeben:

```
ARexx VERSION CPU MPU VIDEO FREQ
```

die einzelnen Komponenten haben folgende Bedeutung:

- VERSION ist die Versionsnummer des Interpreters im Format wie 1.15.
- CPU bezeichnet den Prozessor, der das Programm gegenwärtig ausführt. Möglich sind folgende Werte: 68000, 68010, 68020, 68030, 68040.
- MPU ist entweder NONE, 68881 oder 68882, abhängig davon, ob ein mathematischer Koprozessor vorhanden ist.
- VIDEO steht für NTSC oder PAL.
- FREQ gibt die Netzfrequenz an (60Hz oder 50Hz).

Beispiel:

```
/*Numerische Zeichenfolge ist: "9 0 SCIENTIFIC"*/
PARSE NUMERIC DIGITS FUZZ FORM .
SAY digits     /*9*/
SAY fuzz       /*0*/
SAY form       /*SCIENTIFIC*/
myvar = 1234567890
PARSE VAR myvar 1 a 3 b +2 c 1 d
SAY a
SAY b
SAY c
SAY d
```

Dieses Programm erzeugt folgende Ausgabe:

```
12
34
567890
1234567890
```

PROCEDURE PROCEDURE [EXPOSE Variable [Variable...]]

Der Befehl PROCEDURE wird in einer internen Funktion verwendet, um eine neue Symboltabelle zu erstellen. Dies verhindert, daß die Symbole, die in der Aufrufumgebung definiert sind, bei der Ausführung der Funktion verändert werden. PROCEDURE ist normalerweise die erste Anweisung innerhalb der Funktion, sie wäre aber auch an jeder anderen Stelle zulässig. Werden zwei PROCEDURE-Anweisungen innerhalb einer Funktion ausgeführt, kommt es zu einem Fehler.

Das Unterschlüsselwort EXPOSE bietet ein selektives Verfahren für den Zugriff auf die Symboltabelle der Aufrufebene und für die Weitergabe globaler Variablen an eine Funktion. Die auf das Schlüsselwort EXPOSE folgenden Variablen verweisen auf Symbole in der Symboltabelle der Aufrufebene. Spätere Änderungen dieser Variablen werden in der Umgebung der Aufrufebene berücksichtigt.

Die in der Liste EXPOSE aufgeführten Variablen können Stammsymbole oder zusammengesetzte Symbole enthalten. In diesem Fall ist die Reihenfolge der Variablen von Bedeutung. Die EXPOSE-Liste wird von links nach rechts verarbeitet, zusammengesetzte Symbole werden auf der Basis der für die neue Generation geltenden Werte erweitert. Nehmen wir beispielsweise an, daß der Wert des Symbols J in der vorherigen Generation 123 betrug und daß J in der neuen Generation nicht initialisiert ist. Dann macht die Anweisung PROCEDURE EXPOSE J A.J die Symbole J und A.123 zugänglich, PROCEDURE EXPOSE A.J J dagegen A.J und J. Die Offenlegung eines Stamms bewirkt, daß auch alle möglichen, von diesem Stamm abgeleiteten zusammengesetzten Symbole exponiert werden. Die Anweisung, PROCEDURE EXPOSE A. exponiert also A.I, A.J, A.J.J, A.123 usw. Beispiel:

```
fakult: PROCEDURE       /*Eine rekursive Funktion*/
   ARG i
   IF i = 1
   THEN RETURN 1
   ELSE RETURN i * fakult(i-1)
```

PULL PULL [Schablone] [,Schablone...]

Pull ist die Kurzform des Befehls PARSE UPPER PULL. Er liest eine Zeichenfolge von der Eingabekonsole ein, setzt sie in Großbuchstaben um und analysiert sie mit Hilfe der Schablone. Bei Angabe zusätzlicher Schablonen können mehrere Zeichenfolgen gelesen werden. Der Befehl liest auch dann von der Konsole, wenn keine Schablone zur Verfügung steht. (Schablonen werden in Kapitel 7.1 beschrieben.) Beispiel:

```
PULL Vorname Nachname .      /*Namen lesen*/
```

PUSH PUSH [Ausdruck]

Der Befehl PUSH dient dazu, einen Datenstrom so aufzubereiten, daß er von einer Kommando-Shell oder einem anderen Programm gelesen werden kann. Er fügt ein Zeichen für Zeilenvorschub an das Ergebnis des Ausdrucks an und stellt ihn dann in den Stapel des STDIN-Datenstroms. Stapelzeilen werden nach der Regel "last-in, first-out" (was zuletzt eingegeben wurde, wird zuerst ausgegeben) in den Datenstrom gestellt und können gelesen werden, als ob sie interaktiv eingegeben worden wären. Nach Ausgabe der Befehle:

```
PUSH Zeile 1
PUSH Zeile 2
PUSH Zeile 3
```

würde der Datenstrom in der Reihenfolge Zeile 3, 2, 1 gelesen.

PUSH ermöglicht, daß der STDIN-Datenstrom als privates "Konzeptpapier" zur Vorbereitung von Daten für die spätere Verarbeitung eingesetzt wird. Es könnten z. B. mehrere Dateien mit Einfügung von Begrenzungszeichen verkettet werden, indem einfach die Eingabedateien gelesen, mit PUSH in den Datenstrom gestellt werden und die Begrenzungszeichen bei Bedarf eingefügt werden. Beispiel:

```
DO i=1 to 5
   PUSH 'echo "Zeile 'i'"'
    END
```

QUEUE

QUEUE [Ausdruck]

Der Befehl QUEUE dient dazu, einen Datenstrom so aufzubereiten, daß er von einer Kommando-Shell oder einem anderen Programm gelesen werden kann. Er ist dem Befehl PUSH sehr ähnlich und unterscheidet sich von ihm nur insofern, als die Datenzeilen in der Reihenfolge "first-in, first-out" (Ausgabe in gleicher Reihenfolge wie die Eingabe) in dem STDIN-Datenstrom gestellt werden. In diesem Fall würden die Befehle:

```
QUEUE Zeile 1
QUEUE Zeile 2
QUEUE Zeile 3
```

in der Reihenfolge Zeile 1, 2, 3 gelesen. Die mit QUEUE gestapelten Zeilen gehen stets allen interaktiv eingegebenen Zeilen vor und folgen stets nach den mit PUSH gestapelten Zeilen. Beispiel:

```
DO i=1 to 5
   QUEUE 'echo "Zeile 'i'"'
   END
```

RETURN

RETURN [Ausdruck]

RETURN dient dazu, eine Funktion zu verlassen und die Steuerung an den Punkt zurückzugeben, von dem aus der vorherige Funktionsaufruf erfolgte. Der ausgewertete Ausdruck wird als Resultat der Funktion zurückgegeben. Wird kein Ausdruck angegeben, kann es in der Umgebung der Aufrufebene zu einem Fehler kommen. Funktionen, die von einem Ausdruck aus aufgerufen werden, müssen eine Resultatzeichenfolge zurückgeben. Ist kein Ergebnis verfügbar, kommt es zu einem Fehler. Funktionen, die vom Befehl CALL aufgerufen werden, brauchen kein Ergebnis zurückzugeben.

Wird von der Basisumgebung eines Programms ein RETURN ausgegeben, ist dies kein Fehler, sondern entspricht dem Befehl EXIT (Programm verlassen). Im Abschnitt zum Befehl EXIT wird beschrieben, wie Ergebniszeichenfolgen an eine externe Aufrufebene zurückgeleitet werden. Beispiel:

```
RETURN 6*7    /*Gibt 42 zurück*/
```

SAY SAY [Ausdruck]

Das Ergebnis des ausgewerteten Ausdrucks wird - mit einem angefügten Zeichen für Zeilenvorschub - in die Ausgabekonsole geschrieben. Wird der Ausdruck weggelassen, wird eine leere Zeichenfolge an die Konsole gesendet. Beispiel:

```
SAY 'Die Antwort ist ' Wert
```

SELECT SELECT

SELECT beginnt einen Befehlsblock, der eine oder mehrere WHEN-Klauseln und möglicherweise eine OTHERWISE-Klausel enthält. Auf jede der Klauseln folgt eine bedingte Anweisung. Es wird nur eine der bedingten Anweisungen innerhalb der SELECT-Gruppe ausgeführt. Zunächst werden die einzelnen WHEN-Anweisungen nacheinander ausgeführt, bis eine zutrifft. Trifft keine zu, wird die OTHERWISE-Anweisung ausgeführt. Der SELECT-Bereich muß durch eine END-Anweisung abgeschlossen werden. Beispiel:

```
SELECT
   WHEN i=1 THEN SAY 'eins'
   WHEN i=2 THEN SAY 'zwei'
   OTHERWISE SAY 'andere'
   END
```

SHELL SHELL [Symbol | Zeichenfolge] | [[WERT] [Ausdruck]]

Der Befehl SHELL ist gleichbedeutend mit dem Befehl ADDRESS. Beispiel:

```
SHELL edit    /*Host auf 'EDIT' setzen*/
```

SIGNAL SIGNAL {ON | OFF} Bedingung SIGNAL [WERT] Ausdruck

SIGNAL {ON | OFF} steuert den Status der internen Unterbrechungskennzeichen (interrupt flags). Unterbrechungen ermöglichen es einem Programm, beim Auftreten bestimmter Fehler diese zu erkennen und die Steuerung zu behalten. In diesem Fall muß auf

SIGNAL eines der Schlüsselwörter ON oder OFF und eines der im folgenden beschriebenen Bedingungsschlüsselwörter folgen. Das im Bedingungssymbol angegebene Unterbrechungskennzeichen wird dann auf den angegebenen Status gesetzt. Gültige Signalbedingungen sind:

BREAK_C	Eine Unterbrechung Ctrl-C wurde erkannt.
BREAK_D	Eine Unterbrechung Ctrl-D wurde erkannt.
BREAK_E	Eine Unterbrechung Ctrl-E wurde erkannt.
BREAK_F	Eine Unterbrechung Ctrl-F wurde erkannt.
ERROR	Ein Host-Kommando gab einen Wert ungleich Null zurück.
HALT	Eine externe HALT-Anforderung wurde erkannt.
IOERR	Das E/A-System hat einen Fehler festgestellt.
NOVALUE	Eine nicht initialisierte Variable wurde verwendet.
SYNTAX	Ein Syntax- oder Ausführungsfehler wurde festgestellt.

Die Bedingungsschlüsselwörter werden als Sprungmarken interpretiert, an die die Steuerung übertragen wird, wenn die ausgewählte Bedingung eintritt. Ist z. B. die ERROR-Unterbrechung aktiviert und ein Kommando gibt einen Wert ungleich Null zurück, überträgt ARexx die Steuerung an die Sprungmarke ERROR:. Diese Bedingungssprungmarke muß natürlich im Programm definiert sein. Andernfalls kommt es sofort zu einem SYNTAX-Fehler, und das Programm wird verlassen.

In SIGNAL [WERT] Ausdruck werden die auf SIGNAL folgenden Token als Ausdruck ausgewertet. Es wird eine sofortige Unterbrechung generiert, die die Steuerung an die im Ausdrucksergebnis definierte Sprungmarke übergibt. Der Befehl hat also den Effekt eines "berechneten GOTO".

Sobald eine Unterbrechung eintritt, werden alle derzeit aktiven Steuerbereiche (IF, DO, SELECT, INTERPRET oder interaktiver TRACE) vor der Steuerungsübergabe inaktiviert. Die Übergabe kann also nicht dazu verwendet werden, in den Bereich einer DO-Schleife oder einer anderen Steuerungsstruktur zu springen. Da sich eine SIGNAL-Bedingung nur auf die Steuerungsstrukturen der aktuellen Umgebung auswirkt, ist es gefahrlos möglich, innerhalb einer internen Funktion ein SIGNAL auszulösen. Der Status der Aufrufumgebung wird dadurch nicht berührt.

Die Sondervariable SIGL wird, sobald es zu einer Steuerungsübertragung kommt, auf die aktuelle Zeilennummer gesetzt. Das Programm kann anhand einer Überprüfung von SIGL feststellen, welche Zeile vor der Übertragung der Steuerung ausgeführt wurde. Verursacht eine ERROR- oder SYNTAX-Bedingung eine Unterbrechung, wird die Sondervariable RC auf den Fehlercode gesetzt, der die Unterbrechung auslöste. Im Fall der ERROR-Bedingung gibt dieser Code normalerweise den Schweregrad des Fehlers an. Einzelheiten zu Fehlercodes und Schweregraden finden Sie in Anhang A. Die SYNTAX-Bedingung läßt stets auf einen ARexx-Fehlercode schließen. Beispiel:

```
SIGNAL on error      /*Unterbrechung aktivieren*/
SIGNAL off syntax    /*SYNTAX inaktivieren*/
SIGNAL start         /*Zu START springen*/
```

WHEN WHEN Ausdruck [THEN [;] [bedingte Anweisung]]

Der Befehl WHEN ist dem Befehl IF vergleichbar, ist allerdings nur innerhalb eines SELECT-Bereichs gültig. Alle WHEN-Ausdrücke werden nacheinander ausgewertet und müssen einen booleschen Wert ergeben. Ist das Ergebnis 1, wird die bedingte Anweisung ausgeführt, und die Steuerung geht an die END-Anweisung über, die SELECT beendet. THEN braucht (genau wie beim Befehl IF) nicht Bestandteil derselben Klausel zu sein. Beispiel:

```
SELECT
   WHEN i<j THEN SAY 'kleiner'
   WHEN i=j THEN SAY 'gleich'
   OTHERWISE  SAY 'größer'
   END
```

Kapitel 5
Funktionen

Unter einer Funktion versteht man ein Programm oder eine Gruppe von Anweisungen, das/die ausgeführt wird, wenn der Funktionsname innerhalb eines bestimmten Kontexts aufgerufen wird. Eine Funktion kann Teil eines internen Programms, einer Bibliothek oder eines separaten externen Programms sein. Funktionen sind wichtige Bauteile bei der modularen Programmierung, da sie den Aufbau komplexer Programme aus mehreren kleinen, einfacher zu entwickelnden Modulen ermöglichen.

In diesem Kapitel werden die verschiedenen Funktionsarten vorgestellt und ihre Auswertung beschrieben. Das Kapitel enthält eine alphabetisch geordnete Liste der in der integrierten ARexx-Funktionsbibliothek enthaltenen Funktionen.

5.1 Aufrufen einer Funktion

Innerhalb eines ARexx-Programms ist eine Funktion definiert als ein Symbol oder eine Zeichenfolge, auf das/die unmittelbar eine öffnende runde Klammer folgt. Das Symbol oder die Zeichenkette (interpretiert als Literal) gibt den Funktionsnamen an. Nach der offenen Klammer beginnt die Liste der Argumente. Zwischen den Klammern stehen null oder mehr Argumentausdrücke, getrennt durch Kommas, die die Daten liefern, die an die Funktion übergeben werden.

Gültige Funktionsaufrufe sind:

```
CENTER ('titel',20)
ADDRESS()
'ALLOCMEM'(256*4,1)
```

Alle Argumentausdrücke werden nacheinander ausgewertet, die daraus resultierenden Zeichenfolgen werden als Argumentenliste an die Funktion weitergegeben. Jeder Argumentausdruck (häufig nur ein einzelner Literalwert) kann arithmetische oder Zeichenfolgenoperationen oder auch andere Funktionsaufrufe enthalten. Argumentausdrücke werden in der Reihenfolge von links nach rechts ausgewertet. Funktionen können auch mit dem Befehl CALL aufgerufen werden. Der Befehl CALL (siehe Kapitel 4.2) kann zum Aufrufen einer Funktion eingesetzt werden, die möglicherweise keinen Wert zurückgibt.

5.2 Funktionsarten

Man unterscheidet drei Arten von Funktionen:

- Interne Funktionen — definiert innerhalb des ARexx-Programms.
- Integrierte Funktionen — stammen aus der ARexx-Programmiersprache.
- Funktionsbibliotheken — eine spezielle Amiga-Bibliothek für gemeinsame Benutzung.

5.2.1 Interne Funktion

Eine interne Funktion wird durch eine Sprungmarke innerhalb des Programms gekennzeichnet. Bei Aufruf der internen Funktion erstellt ARexx eine neue Speicherumgebung, damit die Umgebung der vorherigen Aufrufebene unverändert bleibt. Die neue Umgebung übernimmt die Werte ihres Vorgängers, spätere Veränderungen dieser Umgebung haben aber keine Auswirkung auf die vorherige Umgebung.

Folgende Werte werden geschützt bzw. beibehalten:

- Die aktuelle und die vorherige Host-Adresse
- Die Vorgaben für NUMERIC DIGITS, FUZZ und FORM
- Die Ablaufverfolgungsoption, das Sperrungskennzeichen und das interaktive Kennzeichen

- Der Status der durch den Befehl SIGNAL definierten Unterbrechungskennzeichen
- Die aktuelle Zeichenfolge der Eingabeaufforderung gemäß Vorgabe im Befehl OPTIONS PROMPT

Die neue Umgebung erhält nicht automatisch eine neue Symboltabelle, so daß der aufgerufenen Funktion zunächst einmal alle Variablen aus der vorherigen Umgebung zur Verfügung stehen. Der Befehl PROCEDURE kann zum Erstellen einer eigenen Tabelle genutzt werden, wodurch die Symbolwerte der Aufrufebene geschützt werden. PROCEDURE kann auch verwendet werden, um ein und denselben Variablennamen in zwei verschiedenen Bereichen mit zwei unterschiedlichen Werten verwenden zu können.

Die Ausführung der internen Funktion wird fortgesetzt, bis ein RETURN-Befehl ausgeführt wird. An dieser Stelle wird die neue Umgebung wieder inaktiviert, und die Steuerung kehrt zu dem Punkt zurück, an dem der Funktionsaufruf erfolgte. Der mit dem Befehl RETURN angegebene Ausdruck wird ausgewertet und als Funktionsergebnis an die Aufrufebene zurückgeleitet.

5.2.2 Integrierte Funktionen

Als Bestandteil des Sprachsystems bietet ARexx eine umfangreiche Bibliothek vordefinierter Funktionen. Diese Funktionen sind stets verfügbar und auf die Zusammenarbeit mit den internen Datenstrukturen optimal zugeschnitten. Im allgemeinen laufen die integrierten Funktionen wesentlich schneller ab als entsprechende interpretierte Funktionen. Aus diesem Grund wird ihre Verwendung dringend empfohlen.

Manche integrierte Funktionen erstellen und verwalten externe AmigaDOS-Dateien. Auf Dateien wird über einen logischen Namen Bezug genommen. Bei diesem Namen, der einer Datei beim erstmaligen Öffnen zugeordnet wird, wird zwischen Groß- und Kleinschreibung unterschieden. Die ursprünglichen Ein- und Ausgabedatenströme erhalten die Namen STDIN (Standardeingabe) und STDOUT (Standardausgabe). Theoretisch können beliebig viele Dateien gleichzeitig geöffnet sein, allerdings ist der verfügbare Arbeitsspeicher natürlich nicht unendlich groß, so daß Sie

irgendwann an dessen Grenzen stoßen, wenn Sie "unendlich viele" Dateien zu öffnen versuchen. Beim Verlassen des Programms werden alle geöffneten Dateien automatisch geschlossen.

5.2.3 Externe Funktionsbibliotheken

Eine Funktionsbibliothek (engl. shared Library) ist eine Sammlung einer oder mehrerer Funktionen, die als gemeinsam benutzbare Amiga-Bibliothek aufgebaut ist. Die Bibliothek muß in LIBS: gespeichert sein, gleich ob festplatten- oder speicherresident. Festplattenresidente Bibliotheken werden bei Bedarf geladen und geöffnet.

Die Bibliothek muß speziell für die Verwendung durch ARexx zugeschnitten sein. Jede Funktionsbibliothek muß einen Bibliotheksnamen, eine Suchpriorität, eine relative Einstiegsposition und eine Versionsnummer besitzen. Wenn ARexx nach einer Funktion sucht, öffnet der Interpreter jede einzelne Bibliothek und prüft den "query"-Einstiegspunkt. Dieser Einstiegspunkt muß als ganzzahlige Position (z. B. "–30") relativ zur Bibliotheksbasis vorliegen. Der Rückgabewert des Abfrageaufrufs gibt an, ob die gewünschte Funktion gefunden wurde. Wenn ja, wird sie mit den vom Interpreter übergebenen Parametern aufgerufen, und das Funktionsergebnis wird an die Aufrufebene zurückgegeben. Wird sie nicht gefunden, wird ein Fehlercode für "Funktion nicht gefunden" zurückgemeldet, und die Suche wird in der nächsten Bibliothek auf der Liste fortgesetzt. Funktionsbibliotheken werden nach der Überprüfung stets geschlossen, so daß das Betriebssystem bei Bedarf wieder über den Hauptspeicherplatz verfügen kann.

5.2.3.1 Bibliotheksliste

Der residente ARexx-Prozeß verwaltet eine Liste der zur Zeit verfügbaren Funktionsbibliotheken und Funktions-Hosts (s. u.): die sog. Bibliotheksliste. Anwendungsprogramme können bei Bedarf Funktionsbibliotheken hinzufügen oder entfernen.

Die Bibliotheksliste wird als eine nach Prioritätswerten sortierte Listenstruktur verwaltet. Jeder Eintrag besitzt einen ihm zugeordneten Suchprioritätswert von 100 (höchste Priorität) bis –100 (niedrigste Priorität). Einträge können mit einer geeigneten

Priorität hinzugefügt werden, um die Auflösung des Funktionsnamens zu steuern. Bibliotheken mit höheren Prioritätswerten werden zuerst durchsucht. Innerhalb einer vorgegebenen Prioritätsebene werden die zuerst hinzugefügten Bibliotheken auch zuerst durchsucht. Die Prioritätsebenen sind von Bedeutung, wenn z. B. mehrere Bibliotheken identische Definitionen von Funktionsnamen aufweisen, da die in der Suchfolge weiter unten stehenden Funktionen sonst niemals aufgerufen werden können.

5.2.4 Externe Funktions-Hosts

Der einem Funktions-Host zugewiesene Name ist der Name seines öffentlich zugänglichen Message-Ports. Funktionsaufrufe werden in Form eines Nachrichtenpakets an den Host weitergegeben. Danach stellt der jeweilige Host fest, ob er den angegebenen Funktionsnamen erkennt. Die Namensauflösung ist ein interner Host-Vorgang. Funktions-Hosts bieten ein natürliches Gateway-Verfahren zur Implementierung ferner Prozeduraufrufe an andere Maschinen innerhalb eines Netzes. Der residente ARexx-Prozeß ist ein Funktions-Host und wird in der Bibliotheksliste mit einer Priorität von –60 installiert.

5.3 Suchreihenfolge

Funktionseinbindungen in ARexx werden zum Zeitpunkt des Funktionsaufrufs hergestellt. Eine bestimmte Suchreihenfolge wird eingehalten, bis eine Funktion gefunden wird, die mit dem Namenssymbol oder der Zeichenfolge übereinstimmt. Ist die angegebene Funktion nicht auffindbar, hat dies einen Fehler zur Folge, und die Auswertung des Ausdrucks wird beendet. Die vollständige Suchreihenfolge sieht wie folgt aus:

Interne Funktionen

Der Quelltext des Programms wird auf eine Sprungmarke hin untersucht, die dem Funktionsnamen entspricht. Wird eine Entsprechung gefunden, so wird eine neue Speicherumgebung erstellt, und die Steuerung geht auf diese Sprungmarke über.

Integrierte Funktionen

Die Bibliothek mit den integrierten Funktionen wird nach dem angegebenen Namen durchsucht. Alle diese Funktionen werden durch Namen in Großbuchstaben definiert.

Funktionsbibliotheken und Funktions-Hosts

Die verfügbaren Funktionsbibliotheken und Funktions-Hosts werden in der Bibliotheksliste verwaltet, die in der Reihenfolge der Prioritätswerte durchsucht wird, bis die Funktion gefunden oder das Ende der Liste erreicht ist. Funktions-Hosts werden über ein Nachrichtenübertragungsprotokoll aufgerufen, das dem für Kommandos verwendeten Protokoll vergleichbar ist. Sie können als Gateways für ferne Prozeduraufrufe an andere Maschinen innerhalb eines Netzes verwendet werden.

Externe ARexx-Programme

Der letzte Schritt in der Reihenfolge ist die Suche nach einer externen ARexx-Programm-datei. Zu diesem Zweck wird eine Nachricht an den residenten ARexx-Prozeß gesendet. Die Suche beginnt stets im aktuellen Verzeichnis und folgt dann dem gleichen Suchpfad wie der ursprüngliche Aufruf des ARexx-Programms. Beim Namensabgleich spielt Groß- und Kleinschreibung keine Rolle.

Bitte beachten Sie, daß die Prozedur des Funktionsnamenabgleichs bei einigen Suchschritten zwischen Groß- und Kleinschreibung unterscheidet, bei anderen nicht. Ob dies bei der Abgleichsprozedur in einer Funktionsbibliothek oder einem Funktions-Host der Fall ist, liegt im Ermessen des Programmierers. Funktionen, in deren Namen Groß- und Kleinbuchstaben vorkommen, müssen über ein Zeichenfolge-Token aufgerufen werden, da Symbolnamen stets in Großbuchstaben umgewandelt werden.

Die vollständige Suchreihenfolge wird immer dann eingehalten, wenn der Funktionsname durch ein Symbol-Token definiert ist. Die Suche nach internen Funktionen wird dagegen umgangen, wenn der Name in einem Zeichenfolge-Token steht. Dies gibt internen Funktionen die Möglichkeit, die Namen externer Funktionen zu besetzen, wie das folgende Beispiel zeigt:

```
CENTER:              /*internes "CENTER"*/
ARG Zeichenfolge,Lang      /*Argumente holen*/
Lang = MIN(Lang,60) /*Längenwert ggf. ändern*/
return 'CENTER'(Zeichenfolge, Lang)
```

Hier wurde die integrierte Funktion CENTER() nach Änderung des Längenarguments durch eine interne Funktion ersetzt.

5.4 Clip-Liste

Die Clip-Liste ist eine allgemein zugängliche Einrichtung, die als Zwischenspeicher (engl. "clipboard") für die Kommunikation zwischen verschiedenen Programmen dienen kann. Viele Funktionen benutzen diesen Zwischenspeicher zum Abrufen verschiedener Arten von Daten, z. B. vordefinierte Konstanten oder Zeichenfolgen.

Die Clip-Liste verwaltet eine Reihe von Begriffspaaren (Name, Wert), die vielseitig verwendbar sind. (Mit SETCLIP() können Sie neue Begriffspaare in die Liste einfügen.) Jeder Listeneintrag besteht aus einer Namens- und einer Wertzeichenfolge und kann anhand des Namens lokalisiert werden. Im allgemeinen sollten die Namen so gewählt werden, daß sie innerhalb einer Anwendung nur einmal vorkommen, um unbeabsichtigte Namenskonflikte mit anderen Programmen zu vermeiden. Die Anzahl der Listeneinträge ist beliebig.

Eine mögliche Anwendung der Clip-Liste wäre ein Mechanismus zum Laden vordefinierter Konstanten in ein ARexx-Programm. Beispiel:

```
pi=3.14159; e=2.718; sqrt2=1.414  . . .
```

(also eine Reihe von Wertzuweisungen, jeweils getrennt durch Semikolons). Eine solche Zeichenfolge kann anhand ihres Namens

über die integrierte Funktion GETCLIP() abgerufen und danach innerhalb des Programms INTERPRETiert werden. Die Zuordnungsanweisungen innerhalb der Zeichenfolge würden dann die benötigten Konstantendefinitionen bewirken. Beispiel:

```
/*angenommen, die Zeichenfolge "konstanten" ist
verfügbar*/
konstanten = GETCLIP('konstanten')
INTERPRET konstanten        /*. . . Zuordnungen*/
```

Die Zeichenfolge wäre nicht ausschließlich auf Zuweisungsanweisungen beschränkt, sondern könnte auch beliebige gültige ARexx-Anweisungen enthalten. Die Clip-Liste könnte also eine Reihe von Programmen mit Initialisierungen und anderen Verarbeitungsaufgaben versorgen.

Der residente Prozeß unterstützt Hinzufüge- und Löschoperationen zur Verwaltung der Clip-Liste. Bezüglich der Namen in den Begriffspaaren (Name,Wert) wird angenommen, daß sie Groß- und Kleinschreibung enthalten und innerhalb der Liste nicht mehrmals vorkommen. Der Versuch, eine Zeichenfolge mit einem bereits vorhandenen Namen hinzuzufügen, führt lediglich zur Aktualisierung der Wertzeichenfolge. Wenn ein neuer Eintrag in die Liste gestellt wird, werden die Namens- und Wertzeichenfolgen kopiert, so daß das Programm, das einen Eintrag hinzufügt, diese Zeichenfolgen nicht weiter verwalten muß.

In die Clip-Liste gestellte Einträge bleiben verfügbar, bis sie explizit aus der Liste gelöscht werden. Bei Verlassen des residenten Prozesses wird die Clip-Liste automatisch verlassen.

5.5 Liste der integrierten Funktionen

Dieser Abschnitt enthält eine alphabetisch geordnete Liste der integrierten Funktionen. Die Syntax der einzelnen Funktionen steht jeweils rechts vom Funktionsschlüsselwort.

5.5.1 Syntax

Wahlfreie Argumente stehen in eckigen Klammern und haben normalerweise einen Standardwert, der verwendet wird, wenn das Argument nicht explizit angegeben wird. Wird ein Optionsschlüsselwort als Argument angegeben, ist nur das erste Zeichen von Bedeutung. Bei Optionsschlüsselwörtern wird nicht zwischen Groß- und Kleinschreibung unterschieden.

Viele Funktionen erlauben Füllzeichenargumente. Füllzeichen werden eingefügt, um Leerzeichen aufzufüllen bzw. zu erstellen. Bei so einem Füllzeichenargument ist nur das erste Zeichen der Argumentzeichenfolge von Bedeutung. Bei Angabe einer leeren Zeichenfolge wird das standardmäßige Füllzeichen (in der Regel das Leerzeichen) verwendet.

In den folgenden Beispielen bedeutet ein Pfeil (→) "wird ausgewertet als", "ergibt". Der Pfeil wird nicht angezeigt, wenn ein Programm ausgeführt wird. Beispiel:

```
SAY ABS(-5.35) → 5.35
```

Dies bedeutet, daß SAY ABS(–5.35) bei Auswertung 5.35 ergibt.

5.5.2 Alphabetische Liste

ABBREV() ABBREV(Zeichenfolge1,Zeichenfolge2[,Länge])

Diese Funktion gibt einen booleschen Wert zurück, der angibt, ob Zeichenfolge2 eine Abkürzung von Zeichenfolge1 mit einer Länge größer oder gleich der Angabe im Argument "Länge" ist. Standardlänge ist 0, eine leere Zeichenfolge ist also ebenfalls als Abkürzung zulässig. Beispiel:

```
SAY ABBREV('vollname','voll')    → 1
SAY ABBREV('nahezu,'nah',4)    → 0
SAY ABBREV('beliebig','')    →1
```

ABS() ABS(Zahl)

Gibt den absoluten Wert des Arguments "Zahl" zurück. Dies muß ein numerischer Wert sein. Beispiel:

```
SAY ABS(-5.35)    → 5.35
SAY ABS(10)       → 10
```

ADDLIB() ADDLIB(Name,Priorität[,Offset,Version])

Fügt eine Funktionsbibliothek oder einen Funktions-Host in die vom residenten Prozeß verwaltete Bibliotheksliste ein. Das Argument "Name" bezeichnet entweder den Namen einer Funktionsbibliothek oder den einem Funktions-Host zugeordneten, allgemein zugänglichen Message-Port. Bei diesem Namen wird zwischen Groß- und Kleinschreibung unterschieden. Alle angegeben Bibliotheken sollten im Systemverzeichnis LIBS: stehen.

Das Argument "Priorität" bestimmt die Suchpriorität und muß als ganze Zahl im Bereich von 100 bis -100, angegeben werden. Die Argumente "Offset" und "Version" gelten nur für Bibliotheken. Der Offset ist der ganzzahlige Abstand zum "Query"-Einstiegspunkt der Bibliothek. Version ist ebenfalls eine ganze Zahl, die die mindestens erforderliche Version der Bibliothek angibt.

Die Funktion gibt einen booleschen Wert zurück, aus dem hervorgeht, ob die Operation erfolgreich verlief. Beachten Sie bitte, daß eine angegebene Bibliothek zu diesem Zeitpunkt noch nicht geöffnet wird. Entsprechend überprüft ARexx auch noch nicht, ob der Port eines angegebenen Funktions-Hosts geöffnet ist. Beispiel:

```
SAY ADDLIB("rexxsupport.library",0,-30,0) → 1
CALL ADDLIB "EtherNet",-20        /*Ein Gateway*/
```

ADDRESS() ADDRESS()

Gibt die aktuelle Zeichenfolge der Host-Adresse zurück. Die Host-Adresse ist der Message-Port, an den Kommandos gesendet werden. Mit der Funktion SHOW() kann geprüft werden, ob der gewünschte externe Host auch tatsächlich verfügbar ist (siehe auch SHOW()). Beispiel:

```
SAY ADDRESS()    → REXX
```

ARG() ARG([Nummer][,'EXISTS'|'OMITTED'])

ARG() gibt die Anzahl der Argumente zurück, die der aktuellen Umgebung zur Verfügung gestellt wurden. Wird nur das Argument "Nummer" angegeben, wird die entsprechende Argumentzeichenfolge zurückgegeben. Bei Angabe einer Nummer und des Schlüsselworts EXISTS (vorhanden) oder OMITTED (nicht vor-handen) zeigt der boolesche Rückgabewert den Status des entsprechenden Arguments an. Beachten Sie bitte, daß die Überprüfung mit dem Schlüsselwort EXISTS oder OMITTED nicht darauf schließen läßt, ob die betreffende Zeichenfolge einen Nullwert hat, sondern nur, ob überhaupt eine Zeichenfolge angegeben wurde. Beispiel:

```
/*Angenommen, die Argumente sind: ('eins',,10)*/
SAY ARG()    → 3
SAY ARG(1)    → eins
SAY ARG(2,'O')    → 1
```

B2C() B2C(Zeichenfolge)

Wandelt eine Zeichenfolge aus Binärziffern (0,1) in die entsprechende (gepackte) Zeichendarstellung um. Die Umwandlung läuft genau so ab wie bei Angabe des Arguments "Zeichenfolge" als Binärzeichenliteral (z. B. '1010'B). Leerzeichen in der Zeichenfolge sind zulässig, allerdings nur an Bytegrenzen. Diese Funktion eignet sich besonders für die Erstellung von Zeichenfolgen, die als Bitmasken verwendet werden sollen (siehe auch X2C()). Beispiel:

```
SAY B2C('00110011')    → 3
SAY B2C('01100001')    → a
```

BITAND() BITAND(Zeichenfolge1,Zeichenfolge2[,Füllzeichen])

Die Argumente "Zeichenfolge1" und "Zeichenfolge2" werden durch ein logisches UND miteinander verbunden, wobei das Ergebnis so lang ist wie die längere der beiden Operandenzeichenfolgen. Wird ein Füllzeichen angegeben, so wird die kürzere Zeichenfolge rechts damit aufgefüllt. Andernfalls wird die Operation am Ende der kürzeren Zeichenfolge beendet. Der Rest der längeren Zeichenfolge wird an das Ergebnis angehängt. Beispiel:

```
BITAND('0313'x,'FFF0'x)    → '0310'x
```

BITCHG() BITCHG(Zeichenfolge,Bit)

Ändert den Status des angegebenen Bits im Argument "Zeichenfolge". Die Bits sind so numeriert, daß Bit 0 das niederwertigste Bit des am weitesten rechts liegenden Bytes der Zeichenfolge ist. Beispiel:

```
BITCHG('0313'x,4)    → '0303'x
```

BITCLR() BITCLR(Zeichenfolge,Bit)

Mit dieser Funktion wird das angegebene Bit im Argument "Zeichenfolge" gelöscht (d. h. auf Null gesetzt). Die Bits sind so numeriert, daß Bit 0 das niederwertigste Bit des am weitesten rechts liegenden Bytes der Zeichenfolge ist. Beispiel:

```
BITCLR('0313'x,4)  → '0303'x
```

BITCOMP() BITCOMP(Zeichenfolge1,Zeichenfolge2[,Füllzeichen])

Vergleicht die Argumente "Zeichenfolge1" und "Zeichenfolge2" bitweise, beginnend bei Bitnummer 0. Der zurückgegebene Wert ist die Bitnummer des ersten Bits, bei dem sich die Zeichenfolgen unterscheiden, oder –1, falls die Zeichenfolgen identisch sind. Beispiel:

```
BITCOMP('7F'x,'FF'x)    → 7    /*Bit Nr. 7*/
BITCOMP('FF'x,'FF'x)    → -1
```

BITOR() BITOR(Zeichenfolge1,Zeichenfolge2[,Füllzeichen])

Die Argumente "Zeichenfolge1" und "Zeichenfolge2" werden durch ein logisches ODER miteinander verknüpft, wobei das Ergebnis so lang ist wie die längere der beiden Operandenzeichenfolgen. Wird ein Füllzeichen angegeben, so wird die kürzere Zeichenfolge rechts damit aufgefüllt. Andernfalls wird die Operation am Ende der kürzeren Zeichenfolge beendet. Der Rest der längeren Zeichenfolge wird an das Ergebnis angehängt. Beispiel:

```
BITOR('0313'x,'003F'x)    → '033F'x
```

BITSET() BITSET(Zeichenfolge,Bit)

Setzt das angegebene Bit im Argument "Zeichenfolge" auf 1. Die Bits sind so numeriert, daß Bit 0 das niederwertigste Bit des am weitesten rechts liegenden Bytes der Zeichenfolge ist. Beispiel:

```
BITSET('0313'x,2)    → '0317'x
```

BITTST() BITTST(Zeichenfolge,Bit)

Der boolesche Rückgabewert bezeichnet den Status des angegebenen Bits im Argument "Zeichenfolge". Die Bits sind so numeriert, daß Bit 0 das niederwertigste Bit des am weitesten rechts liegenden Bytes der Zeichenfolge ist. Beispiel:

```
BITTST('0313'x,4)    → 1
```

BITXOR() BITXOR(Zeichenfolge1,Zeichenfolge2[,Füllzeichen])

Die Argumente "Zeichenfolge1" und "Zeichenfolge2" werden durch ein logisches Exklusiv-ODER miteinander verknüpft, wobei das Ergebnis so lang ist wie die längere der beiden Operandenzeichenfolgen. Wird ein Füllzeichen angegeben, so wird die kürzere Zeichenfolge rechts damit aufgefüllt. Andernfalls wird die Operation am Ende der kürzeren Zeichenfolge beendet, und der Rest der längeren Zeichenfolge wird an das Ergebnis angehängt. Beispiel:

```
BITXOR('0313'x,'001F'x)    → '030C'X
```

C2B() C2B(Zeichenfolge)

Wandelt die Zeichenfolge aus Textzeichen in die entsprechende Zeichenfolge aus Binärziffern um (siehe auch C2X()). Beispiel:

```
SAY C2B('abc')    → 011000010110001001100011
```

C2D() C2D(Zeichenfolge[,n])

Wandelt das Argument "Zeichenfolge" von normalen Textzeichen in die entsprechende Dezimalzahl um (ausgedrückt in ASCII-Ziffern, als 0-9). Wird n angegeben, so wird die Textzeichenfolge als in n Byte angegebene Zahl betrachtet. Die Zeichenfolge wird abgeschnitten oder links mit Nullen aufgefüllt. Das Vorzeichenbit wird für die Umwandlung erweitert. Beispiel:

```
SAY C2D('0020'x)    → 32
SAY C2D('FFFF ffff'x)    → -1
SAY C2D('FF0100'x,2)    → 256
```

C2X() C2X(Zeichenfolge)

Wandelt das Argument "Zeichenfolge" von normalen Textzeichen in die entsprechende Hexadezimalzahl um (ausgedrückt in den ASCII-Zeichen 0-9 und A-F (siehe auch C2B()). Beispiel:

```
SAY C2X('abc')    → 616263
```

CENTER() CENTER(Zeichenfolge,Länge[,Füllzeichen])
CENTRE() CENTRE(Zeichenfolge,Länge[,Füllzeichen])

Zentriert das Argument "Zeichenfolge" innerhalb einer Zeichenfolge der angegebenen Länge. Ist der Wert "Länge" größer als die Länge der Zeichenfolge, werden Füll- oder Leerzeichen nach Bedarf eingefügt. Beispiel:

```
SAY CENTER('abc',6)          → ' abc  '
SAY CENTER('abc',6,'+')      → '+abc++'
SAY CENTER('123456',3)       → '234'
```

CLOSE() CLOSE(Datei)

Schließt die Datei, deren logischer Name angegeben wurde. War die Datei geöffnet, zeigt der boolesche Wert 1 die erfolgreiche Durchführung der Operation an. Beispiel:

```
SAY CLOSE('input')    → 1
```

COMPARE() COMPARE(Zeichenfolge1,Zeichenfolge2[,Füllz.])

Vergleicht zwei Zeichenfolgen und gibt den Indexwert der ersten Position zurück, an der sich die Zeichenfolgen unterscheiden (oder 0, wenn die Zeichenfolgen identisch sind). Die kürzere Zeichenfolge wird bei Bedarf rechts mit dem angegebenen Füllzeichen oder mit Leerzeichen aufgefüllt. Beispiel:

```
SAY COMPARE('abcde','abcce')     → 4
SAY COMPARE('abcde','abcde')     → 0
SAY COMPARE('abc++','abc+-','+')    → 5
```

COMPRESS() COMPRESS(Zeichenfolge[,Liste])

Wird das Argument "Liste" weggelassen, entfernt die Funktion voran- oder nachgestellte sowie eingebettete Leerzeichen aus dem Argument "Zeichenfolge". Wird das wahlfreie Argument "Liste" (auch eine Zeichenfolge) dagegen angegeben, legt es die aus der Zeichenfolge zu entfernenden Zeichen fest. Beispiel:

```
SAY COMPRESS('  wie viel  ')    → wieviel
SAY COMPRESS('++12-34-+','+-')    → 1234
```

COPIES() COPIES(Zeichenfolge,Zahl)

Erstellt eine neue Zeichenfolge, indem die angegebene Anzahl Kopien der Originalzeichenfolge aneinandergefügt wird. Als Zahl kann auch Null angegeben werden. In diesem Fall wird eine leere Zeichenfolge zurückgegeben. Beispiel:

```
SAY COPIES('abc',3)    → abcabcabc
```

D2C() D2C(Zahl)

Erstellt eine Zeichenfolge, deren Wert die binäre (gepackte) Darstellung der angegebenen Dezimalzahl ist. Beispiel:

```
D2C(65)    → A
```

D2X() D2X(Zahl[,Ziffern])

Wandelt eine Dezimalzahl in eine Hexadezimalzahl um. Beispiel:

```
D2X(31)    → 1F
```

DATE() DATE([Option][,Datum][,Format])

Gibt das aktuelle Datum im angegebenen Format zurück. Die Standardoption ('NORMAL') gibt das Datum im Format TT MMM JJJJ zurück, z. B. 20 APR 1992. Folgende Optionen werden erkannt:

BASEDATE	Die Anzahl der Tage seit dem 1. Januar 0001
CENTURY	Die Anzahl der Tage seit dem 1. Januar dieses Jahrhunderts
DAYS	Die Anzahl der Tage seit dem 1. Januar des aktuellen Jahres
EUROPEAN	Das Datum im Format TT/MM/JJ
INTERNAL	Interne Systemtage
JULIAN	Das Datum im Format JJTTT
MONTH	Der aktuelle Monat (in Groß- und Kleinschreibung)
NORMAL	Das Datum im Format TT MMM JJJJ
ORDERED	Das Datum im Format JJ/MM/TT
SORTED	Das Datum im Format JJJJMMTT
USA	Das Datum im Format MM/TT/JJ
WEEKDAY	Der Wochentag (in Groß- und Kleinschreibung)

Diese Optionen können abgekürzt werden. Es genügt die Angabe des jeweils ersten Zeichens.

Die Funktion DATE() akzeptiert auch wahlfreie zweite und dritte Argumente, mit denen das Datum in Form der internen Systemtage oder in der 'sortierten' Form JJJJMMTT angegeben werden kann. Das zweite Argument gibt entweder Systemtage (Standardwert) oder ein sortiertes Datumsformat an. Das dritte Argument bestimmt, welches dieser Formate verwendet wird und kann entweder 'I' oder 'S' lauten. Das aktuelle Datum in Systemtagen kann mit der Angabe DATE('INTERNAL') abgerufen werden. Beispiel:

```
SAY DATE()              → 14 Jul 1992
SAY DATE('M')           → July
SAY DATE(S)             → 19920714
SAY DATE('S',DATE('I')+21)    → 19920804
SAY DATE('W',19890609,'S')    → Friday
```

DATATYPE() DATATYPE(Zeichenfolge[,Option])

Wird nur das Argument "Zeichenfolge" angegeben, testet DATATYPE(), ob der Zeichenfolgenparameter eine gültige Zahl ist und gibt entweder NUM oder CHAR zurück. Ist auch ein Optionsschlüsselwort angegeben, gibt der boolesche Rückgabewert Aufschluß darüber, ob die Zeichenfolge dem angegebenen Typ entspricht. Folgende Optionsschlüsselwörter werden erkannt:

ALPHANUMERIC	Alphabetische (A-Z, a-z, Umlaute, ß) und numerische (0-9) Zeichen
BINARY	Binäre Ziffernfolge
LOWERCASE	Kleinbuchstaben (a-z, ä, ö, ü)
MIXED	Groß- und Kleinbuchstaben
NUMERIC	Gültige Zahlen
SYMBOL	Gültige REXX-Symbole
UPPER	Großbuchstaben (A-Z, Ä, Ö, Ü, ß)
WHOLE	Ganze Zahlen
X	Hexadezimale Ziffernfolgen

Beispiel:

```
SAY DATATYPE('123')    → NUM
SAY DATATYPE('1a f2','X')    → 1
SAY DATATYPE('aBcde','L')    → 0
```

DELSTR() DELSTR(Zeichenfolge,n[,Länge])

Löscht den Teil des Arguments "Zeichenfolge", der mit dem n-ten Zeichen beginnt und aus der mit "Länge" angegebenen Anzahl Zeichen besteht. Standardlänge ist die verbleibende Länge der Zeichenfolge. Beispiel:

```
SAY DELSTR('123456',2,3)    → 156
```

DELWORD() DELWORD(Zeichenfolge,n[,Länge])

Löscht den Teil des Arguments "Zeichenfolge", der mit dem n-ten Wort beginnt und aus der mit "Länge" angegebenen Anzahl Wörter besteht. Standardlänge ist der komplette Rest der Zeichenfolge. Zur gekürzten Zeichenfolge gehören auch alle dem letzten Wort nachgestellten Leerzeichen. Beispiel:

```
SAY DELWORD('Sag mir was Nettes',2,2)    → 'Sag
Nettes'
SAY DELWORD('eins zwei drei',3)    → 'eins zwei '
```

DIGITS() DIGITS()

Gibt die aktuelle Vorgabe für NUMERIC DIGITS zurück. Beispiel:

```
NUMERIC DIGITS 6
SAY DIGITS()     → 6
```

EOF() EOF(Datei)

Prüft die angegebene logische Datei und gibt den booleschen Wert 1 (Wahr) zurück, wenn das Dateiende erreicht ist, andernfalls 0 (Falsch). Beispiel:

```
SAY EOF(Eindatei)    → 1
```

ERRORTEXT() ERRORTEXT(n)

Gibt den dem angegebenen ARexx-Fehlercode zugehörigen Fehlermeldungstext zurück. Ist die Zahl kein gültiger Fehlercode, wird eine leere Zeichenfolge zurückgegeben. Beispiel:

```
SAY ERRORTEXT(41)    → Invalid expression
                       (Ungültiger Ausdruck)
```

EXISTS() EXISTS(Dateiname)

Testet, ob eine externe Datei mit dem angegebenen Namen existiert. Der Name kann auch Geräte- und Verzeichnisangaben enthalten. Beispiel:

```
SAY EXISTS('SYS:C/ED')    → 1
```

EXPORT() EXPORT(Adresse[,Zeichenfolge][,Länge][,Füllzeichen])

Kopiert Daten aus der wahlfreien Zeichenfolge in den zuvor reservierten Speicherbereich. Dieser Speicherbereich muß in Form einer 4-Byte-Adresse angegeben werden. Das Längenargument bezeichnet die maximale Anzahl zu kopierender Zeichen. Standardwert ist die Länge der Zeichenfolge. Ist die angegebene Länge größer als die der Zeichenfolge, wird der restliche Bereich mit dem Füllzeichen oder mit Nullzeichen ('00'x) aufgefüllt. Der Rückgabewert gibt die Anzahl der kopierten Zeichen an.

Achtung **Jeder beliebige Speicherbereich kann überschrieben werden, was eventuell einen Systemabsturz zur Folge hat. Taskwechsel werden während des Kopiervorgangs unterbunden, so daß die Systemleistung beim Kopieren langer Zeichenfolgen möglicherweise sinkt.**

Siehe auch IMPORT() und STORAGE(). Beispiel:

```
anzahl = EXPORT('0004 0000'x,'Die Antwort')
```

FORM() FORM()

Gibt die aktuelle Vorgabe für NUMERIC FORM zurück. Beispiel:

```
NUMERIC FORM SCIENTIFIC
SAY FORM()    → SCIENTIFIC
```

FIND() FIND(Zeichenfolge,Phrase)

Die Funktion FIND() sucht eine Phrase aus mehreren Wörtern in einer Wortfolge und gibt die Wortnummer des Anfangs der Übereinstimmung zurück. Beispiel:

```
SAY FIND('Nun ist es Zeit','ist es')    → 2
```

FREESPACE() FREESPACE (Adresse, Länge)

Gibt einen Speicherblock mit der vorgegebenen Länge an den internen Pool des Interpreters zurück. Das Argument "Adresse" muß eine 4 Byte lange Zeichenfolge sein, die man durch einen vorherigen Aufruf der internen Zuordnungsfunktion GETSPACE() erhalten hat. Es ist nicht immer erforderlich, intern reservierten Speicherplatz freizugeben, da dieser bei Beendigung des Programms dem System sowieso wieder zu Verfügung gestellt wird. Wurde jedoch ein sehr großer Speicherblock reserviert, kann seine Rückgabe an den Pool möglichen Speicherplatz-Problemen vorbeugen. Der Rückgabewert ist ein boolesches Erfolgskennzeichen. Vergleichen Sie dazu auch die Funktion GETSPACE():

Zum Beispiel:

```
SAY FREESPACE ('00042000'x,32) → 1
```

Wird die Funktion FREESPACE() ohne Argumente aufgerufen, dann wird der im internen Pool des Interpreters verfügbare Speicherplatz zurückgegeben.

FUZZ() FUZZ()

Gibt die aktuelle Vorgabe für NUMERIC FUZZ zurück. Beispiel:

```
NUMERIC FUZZ 3
SAY FUZZ()    → 3
```

GETCLIP() GETCLIP(Name)

Durchsucht die Clip-Liste nach einem Eintrag, der dem angegebenen Argument "Name" entspricht, und gibt die diesem Parameter zugeordnete Wertzeichenfolge zurück. Bei diesem Namensabgleich wird nicht zwischen Groß- und Kleinschreibung unterschieden. Wenn der Name in der Clip-Liste nicht gefunden wird, wird eine leere Zeichenfolge zurückgegeben (siehe auch SETCLIP()). Beispiel:

```
/*Angenommen 'Zahlen' enthält 'PI=3.14159'*/
SAY GETCLIP('Zahlen')    → PI=3.14159
```

GETSPACE() GETSPACE(Länge)

Reserviert einen Speicherblock der angegebenen Länge aus dem internen Pool des Interpreters. Der Rückgabewert ist die 4-Byte-Adresse des reservierten Blocks, der weder gelöscht noch auf andere Weise initialisiert wird. Sobald das ARexx-Programm beendet wird, wird der interne Speicher automatisch wieder dem System zur Verfügung gestellt. Daher sollte die Funktion nicht verwendet werden, um Speicherplatz für externe Programme zu reservieren. Die Unterstützungsbibliothek (REXXSupport.Library) enthält die Funktion ALLOCMEM(), die Speicherplatz aus der Liste für freien Systemspeicher reserviert (siehe auch FREESPACE()). Beispiel:

```
SAY C2X(GETSPACE(32))    → '0003BF40'x
```

HASH() HASH(Zeichenfolge)

Gibt das Hash-Attribut einer Zeichenfolge als Dezimalzahl zurück und aktualisiert den internen Hash-Wert der Zeichenfolge. Beispiel:

```
SAY HASH('1')    → 49
```

IMPORT() IMPORT(Adresse[,Länge])

Diese Funktion erstellt eine Zeichenfolge, indem sie Daten ab einer angegebenen 4-Byte-Adresse kopiert. Wird kein Argument "Länge" angegeben, endet der Kopiervorgang, sobald ein Byte mit dem Wert Null gefunden wird (siehe auch EXPORT()). Beispiel:

```
extval = IMPORT('0004 0000'x,8)
```

INDEX() INDEX(Zeichenfolge,Muster[,Start])

Sucht nach dem ersten Vorkommen des Arguments "Muster" im Argument "Zeichenfolge", beginnend an der angegebenen Startposition. Standardmäßige Startposition ist 1. Der Rückgabewert ist der Index der mit dem Muster übereinstimmenden Zeichenfolge oder 0, wenn das Muster nicht gefunden wurde. Beispiel:

```
SAY INDEX("123456","23")    → 2
SAY INDEX("123456","77")    → 0
SAY INDEX("123123","23",3) → 5
```

INSERT() INSERT(neu,alt[,Start][,Länge][,Füllzeichen])

Fügt nach der angegebenen Startposition die neue Zeichenfolge in die alte Zeichenfolge ein. Standardmäßige Startposition ist 0. Die neue Zeichenfolge wird je nach Bedarf auf die angegebene Länge abgeschnitten oder aufgefüllt (mit dem angegebenen Füllzeichen oder mit Leerzeichen). Liegt die Startposition hinter dem Ende der alten Zeichenfolge, wird die alte Zeichenfolge rechts aufgefüllt. Beispiel:

```
SAY INSERT('ab','12345')    → ab12345
SAY INSERT('123','++',3,5,'-')    → ++-123--
```

LASTPOS() LASTPOS(Muster,Zeichenfolge[,Start])

Sucht rückwärts nach dem ersten Vorkommen des angegebenen Musters im Argument "Zeichenfolge", beginnend an der angegebenen Startposition. Standardmäßige Startposition ist das Ende der Zeichenfolge. Der Rückgabewert ist der Index der mit dem Muster übereinstimmenden Zeichenfolge oder 0, wenn das Muster nicht gefunden wurde. Beispiel:

```
SAY LASTPOS('2','1234')    → 2
SAY LASTPOS('2','1234234')    → 5
SAY LASTPOS('2','123234',3)    → 2
SAY LASTPOS('2','13579')    → 0
```

LEFT() LEFT(Zeichenfolge,Länge[,Füllzeichen])

Diese Funktion gibt die am weitesten links stehende Teilzeichenfolge aus dem angegebenen Argument "Zeichenfolge" mit der angegebenen Länge zurück. Ist die Teilzeichenfolge kürzer als die angeforderte Länge, wird sie rechts mit dem angegebenen Füllzeichen oder mit Leerzeichen aufgefüllt. Beispiel:

```
SAY LEFT('123456',3)    → 123
SAY LEFT('123456',8,'+')    → 123456++
```

LENGTH() LENGTH(Zeichenfolge)

Gibt die Länge der Zeichenfolge zurück. Beispiel:

```
SAY LENGTH('drei')    → 4
```

LINES() LINES(Datei)

Gibt an, wieviele Zeilen einer logischen Datei in der Warteschlange stehen oder bereits eingegeben wurden, wobei sich die logische Datei auf einen interaktiven Datenstrom beziehen muß. Die Zeilenzahl ergibt sich als Sekundärergebnis des Aufrufs WaitForChar(). Beispiel:

```
PUSH 'eine Zeile'
PUSH 'noch eine'
SAY LINES(STDIN)    → 2
```

MAX() MAX(Zahl,Zahl[,Zahl, . . .])

Gibt den höchsten Wert der angegebenen Argumente zurück. Alle Argumente müssen numerisch sein, und es müssen mindestens zwei Parameter angegeben werden. Beispiel:

```
SAY MAX(2.1,3,-1)    → 3
```

MIN() MIN(Zahl,Zahl[,Zahl, . . .])

Gibt den niedrigsten Wert der angegebenen Argumente zurück. Alle Argumente müssen numerisch sein, und es müssen mindestens zwei Parameter angegeben werden. Beispiel:

```
SAY MIN(2.1,3,-1)    → -1
```

OPEN() OPEN(Datei,Dateiname[,'APPEND' | 'READ' | 'WRITE'])

Öffnet eine externe Datei für die angegebene Operation. Das Zeichenfolgenargument "Datei" definiert den logischen Namen, unter dem auf die Datei verwiesen wird. Der "Dateiname" ist der DOS-Name der Datei und kann sowohl Geräte- als auch Verzeichnisangaben enthalten. Die Funktion gibt einen booleschen Wert zurück, der darüber Aufschluß gibt, ob die Operation erfolgreich ausgeführt wurde. Die Anzahl gleichzeitig geöffneter Dateien ist theoretisch unbegrenzt. Bei Verlassen des Programms werden alle

geöffneten Dateien automatisch geschlossen (siehe auch CLOSE(), READ() und WRITE()). Beispiel:

```
SAY OPEN('MeinCon','CON:160/50/320/100/MeinCon/cds')
→ 1
SAY OPEN('Ausgabe','ram:temp','W')    → 1
```

OVERLAY() OVERLAY(neu,alt[,Start][,Länge][,Füllzeichen])

Überlagert die alte Zeichenfolge ab der angegebenen Startposition mit der neuen Zeichenfolge. Die Startposition muß ein positiver Wert sein. Standardmäßige Startposition ist 1. Die neue Zeichenfolge wird bei Bedarf abgeschnitten oder mit dem angegebenen Füllzeichen oder mit Leerzeichen bis zur angegebenen Länge aufgefüllt. Liegt die Startposition hinter dem Ende der alten Zeichenfolge, wird die alte Zeichenfolge rechts aufgefüllt. Beispiel:

```
SAY OVERLAY('bb','abcd')    → bbcd
SAY OVERLAY('4','123',5,5,'-')  → 123-4----
```

POS() POS(Muster,Zeichenfolge[,Start])

Sucht nach dem ersten Vorkommen des Arguments "Muster" im Argument "Zeichenfolge", beginnend an der im Argument "Start" angegebenen Position. Standardmäßige Startposition ist 1. Der Rückgabewert ist der Index der mit dem Muster übereinstimmenden Zeichenfolge oder 0, wenn das Muster nicht gefunden wurde. Beispiel:

```
SAY POS('23','123234')        → 2
SAY POS('77','123234')        → 0
SAY POS('23','123234',3)      → 4
```

PRAGMA() PRAGMA(Option[,Wert])

Diese Funktion ermöglicht einem Programm die Änderung verschiedener Attribute, die sich auf die Systemumgebung beziehen, in der das Programm ausgeführt wird. Das Argument "Option" ist ein Schlüsselwort, das ein Umgebungsattribut angibt. Das Argument "Wert" bezeichnet den neu zu setzenden Attributwert. Der von der Funktion zurückgegebene Wert hängt vom gewählten Attribut ab. Einige Attribute geben den zuvor gültigen Wert zurück, andere lediglich einen booleschen Wert, der über Erfolg oder Scheitern der Operation Auskunft gibt.

Die zur Zeit definierten Optionsschlüsselwörter sind:

DIRECTORY Gibt ein neues aktuelles Verzeichnis an. Das aktuelle Verzeichnis gilt als Bezug für Dateinamen, die keine explizite Geräteangabe enthalten. Rückgabewert ist der Name des alten Verzeichnisses. PRAGMA('D') entspricht PRAGMA('D',""). Damit wird der Pfadname des aktuellen Verzeichnisses zurückgegeben, ohne das Verzeichnis zu wechseln.

PRIORITY Gibt eine neue Task-Priorität an. Der Prioritätswert muß eine ganze Zahl im Bereich von –128 bis 127 sein, in der Praxis ist dieser Bereich allerdings wesentlich enger gefaßt. ARexx-Programme sollten in keinem Fall eine höhere Priorität erhalten als der residente Prozeß, der zur Zeit mit Priorität 4 ausgeführt wird. Der Rückgabewert ist die zuvor gültige Prioritätsstufe.

ID Gibt die Task-ID (die Adresse des Task-Blocks) in Form einer hexadezimalen 8-Hexziffern-Zeichenfolge zurück. Die Task-ID ist eine eindeutige Kennung für den jeweiligen ARexx-Aufruf und kann verwendet werden, um aus diesem einen eindeutigen (Datei)Namen zu generieren.

STACK Gibt eine neue Stackgröße für das aktuelle ARexx-Programm an. Wenn eine neue Stackgröße festgelegt wird, wird der alte Wert zurückgegeben.

Die zur Zeit implementierten Optionen sind:

PRAGMA('W',{'NULL'| 'WORKBENCH'})	Steuert das Feld WindowPtr der Task. Die Einstellung auf 'NULL' unterdrückt alle Dialogfenster, die ansonsten durch einen DOS-Aufruf generiert werden könnten.
PRAGMA('*'[,Name])	Definiert den angegebenen logischen Namen als aktuellen Konsolen-Handler ("*"). Damit kann der Benutzer zwei Datenströme innerhalb eines Fensters öffnen. Wird der Name weggelassen, erhält der Konsolen-Handler den Namen des Klientenprozesses.

```
SAY PRAGMA('D','DF0:C')    → Extras
SAY PRAGMA('D','DF1:C')    → Workbench:C
SAY PRAGMA('PRIORITY',-5)    → 0
SAY PRAGMA('ID')    → 00221ABC
CALL PRAGMA '*',STDOUT
SAY PRAGMA("STACK",8092)    → 4000
```

RANDOM() RANDOM([MIN][,MAX][,Ausgangszahl])

Gibt eine ganzzahlige Pseudo-Zufallszahl aus dem mit den Argumenten MIN und MAX festgelegten ganzzahligen Wertebereich zurück. Die Standardwerte sind 0 (Minimum) und 999 (Maximum). Das Intervall Max-Min muß kleiner oder gleich 1000 sein. Ist ein größerer Wertebereich für die Zufallszahlen erforderlich, können die Werte der Funktion RANDU() entsprechend skaliert und umgewandelt werden. Das Argument "Ausgangszahl" kann angegeben werden, um den internen Status des Zufallszahlengenerators zu initialisieren (siehe auch RANDU()). Beispiel:

```
dieserwurf = RANDOM(1,6)    /*Könnte 1 sein*/
neuerwurf = RANDOM(1,6) /*2 Augen?*/
```

RANDU() RANDU([Ausgangszahl])

Gibt eine gleichmäßig verteilte Pseudo-Zufallszahl im Bereich von 0 bis 1 zurück. Die Anzahl der die Genauigkeit bestimmenden Dezimalstellen ist stets gleich der Vorgabe für Numeric Digits. Mit der Auswahl geeigneter Skalierungs- und Umsetzungswerte kann

RANDU() verwendet werden, um Pseudo-Zufallszahlen in einem beliebigen Intervall zu generieren.

Das wahlfreie Argument "Ausgangszahl" kann angegeben werden, um den internen Status des Zufallszahlengenerators zu initialisieren (siehe auch RANDOM()). Beispiel:

```
erstversuch = RANDU()    /*0.371902021?*/
NUMERIC DIGITS 3
nochmal = RANDU()    /*0.873?*/
```

READCH() READCH(Datei,Länge)

Liest die angegebene Anzahl Zeichen aus der angegebenen logischen Datei in eine Zeichenfolge ein. Die Länge der zurückgegebenen Zeichenfolge entspricht der Anzahl tatsächlich gelesener Zeichen und kann auch kleiner sein als die angeforderte Länge, z. B. wenn das Dateiende erreicht wurde (siehe auch READLN()). Beispiel:

```
zeile = READCH('Eingabe',10)
```

READLN() READLN(Datei)

Liest Zeichen aus der angegebenen logischen Datei in eine Zeichenfolge ein, bis ein Zeichen für Zeilenvorschub festgestellt wird, also eine komplette Zeile. Die zurückgegebene Zeichenfolge enthält das Zeichen für Zeilenvorschub nicht (siehe auch READCH()). Beispiel:

```
zeile = READLN('MeinDatei')
```

REMLIB() REMLIB(Name)

Entfernt einen Eintrag mit dem angegebenen Namen aus der Bibliotheksliste, die vom residenten Prozeß verwaltet wird. Wenn der Eintrag gefunden und erfolgreich entfernt wurde, wird der boolesche Wert 1 zurückgegeben. Diese Funktion unterscheidet nicht zwischen Funktionsbibliotheken und Funktions-Hosts, sondern entfernt einfach den angegebenen Eintrag (siehe auch ADDLIB()). Beispiel:

```
SAY REMLIB('MeineLibrary.Library')   → 1
```

REVERSE() REVERSE(Zeichenfolge)

Kehrt die Reihenfolge der Zeichen in der Zeichenfolge um. Beispiel:

```
SAY REVERSE('?thcin muraW')   → Warum nicht?
```

RIGHT() RIGHT(Zeichenfolge,Länge[,Füllzeichen])

Diese Funktion gibt die am weitesten rechts stehende Teilzeichenfolge aus dem angegebenen Argument "Zeichenfolge" mit der angegebenen Länge zurück. Ist die Teilzeichenfolge kürzer als die angeforderte Länge, wird sie links mit dem angegebenen Füllzeichen oder mit Leerzeichen aufgefüllt. Beispiel:

```
SAY RIGHT('123456',4)    → 3456
SAY RIGHT('123456',8,'+')    → ++123456
```

SEEK() SEEK(Datei,Offset[,'BEGIN'|'CURRENT'|'END'])

Geht zu einer neuen Position in der angegebenen logischen Datei. Die neue Position wird als Abstand (Offset) bezogen auf eine Ankerposition (drittes Argument) angegeben. Standardmäßige Ankerposition ist die aktuelle Position (CURRENT). Der Rückgabewert ist die neue Position bezogen auf den Dateianfang. Beispiel:

```
SAY SEEK('Eingabe',10,'B')    → 10
SAY SEEK('Eingabe',0,'E') → 356 /*Dateilänge*/
```

SETCLIP() SETCLIP(Name[,Wert])

Fügt ein Name-Wert-Paar in die Clip-Liste ein, die vom residenten Prozeß verwaltet wird. Wenn ein Eintrag mit diesem Namen bereits existiert, wird sein Wert entsprechend der angegebenen Wertzeichenfolge aktualisiert. Einträge können durch Angabe eines Nullwertes (gar kein Argument) entfernt werden. Die Funktion gibt einen booleschen Wert zurück, der Aufschluß darüber gibt, ob die Operation erfolgreich ausgeführt wurde. Beispiel:

```
SAY SETCLIP('Pfad','DF0:s')    → 1
SAY SETCLIP('Pfad')    → 1
```

SHOW() SHOW(Option[,Name][,Füllzeichen])

Gibt die eingetragenen Namen in der Ressourcenliste "Option" zurück oder überprüft, ob ein Eintrag mit dem angegebenen Namen vorliegt. Die zur Zeit implementierten Optionsschlüsselwörter sind:

CLIP	Überprüft die Namen in der Clip-Liste.
FILES	Überprüft die Namen der zur Zeit geöffneten logischen Dateien.
LIBRARIES	Überprüft die Namen in der Bibliotheksliste, die weder Funktionsbibliotheken noch Funktions-Hosts bezeichnen.
PORTS	Überprüft die Namen in der Liste der System-Ports.

Wird das Argument "Name" weggelassen, gibt die Funktion eine Zeichenfolge mit den Ressourcennamen zurück, jeweils getrennt durch Leerzeichen oder (falls eines definiert wurde) das angegebene Füllzeichen. Wird das Argument "Name" angegeben, gibt der zurückgegebene boolesche Wert darüber Aufschluß, ob der Name in der Ressourcenliste gefunden wurde. Bei diesen Namenseinträgen wird zwischen Groß- und Kleinschreibung unterschieden.

SIGN() SIGN(Zahl)

Gibt 1 zurück, wenn das Argument "Zahl" positiv oder Null ist, und –1, wenn die Zahl negativ ist. Das Argument muß numerisch sein. Beispiel:

```
SAY SIGN(12)    → 1
SAY SIGN(-33)   → -1
```

SOURCELINE() SOURCELINE([Zeile])

Gibt den Text der angegebenen Zeile des zur Zeit laufenden ARexx-Quellprogramms zurück. Wenn das Argument "Zeile" weggelassen wird, gibt die Funktion die Gesamtzeilenzahl der Datei zurück. Diese Funktion wird häufig verwendet, um Hilfeinformationen in ein Programm einzufügen. Beispiel:

```
/*Ein einfaches Testprogramm*/
SAY SOURCELINE()    → 3
SAY SOURCELINE(1) → /*Ein einfaches Testprogramm*/
```

SPACE() SPACE(Zeichenfolge,n[,Füllzeichen])

Formatiert das Argument "Zeichenfolge" so um, daß zwischen jeweils 2 Wörtern n Leerzeichen stehen. Wenn das Füllzeichen angegeben wird, wird es anstelle von Leerzeichen als Trennzeichen verwendet. Wird für n 0 angegeben, werden alle Leerzeichen aus der Zeichenfolge entfernt. Beispiel:

```
SAY SPACE('Nun ist es Zeit',3)
    → 'Nun   ist   es   Zeit'
SAY SPACE('Nun ist es Zeit',0)
    → 'NunistesZeit'
SAY SPACE('1 2 3',1,'+')    → '1+2+3'
```

STORAGE() STORAGE([Adresse][,Zeichenfolge][,Länge][,Füllz.])

Die Funktion STORAGE() ohne Argumente gibt den verfügbaren Systemspeicher zurück. Wird das Argument "Adresse" angegeben, muß dies in Form einer 4-Byte-Zeichenfolge geschehen. Die Funktion kopiert Daten aus der (wahlfrei anzugebenden) Zeichenfolge an die angegebene Speicheradresse. Der Parameter "Länge" bezeichnet die maximale Anzahl zu kopierender Bytes und hat standardmäßig die Länge der Zeichenfolge. Ist die angegebene Länge größer als die Zeichenfolge, wird der verbleibende Rest mit dem Füllzeichen oder mit Nullbytes ('00'x) aufgefüllt.

Der Rückgabewert bezeichnet den vorherigen Inhalt des Speicherbereichs. Dieser kann in einem späteren Aufruf verwendet werden, um den ursprünglichen Inhalt wiederherzustellen (siehe auch EXPORT()).

Vorsicht **Jeder beliebige Speicherbereich kann überschrieben werden, was eventuell einen Systemabsturz zur Folge hat. Taskwechsel werden während des Kopiervorgangs unterbunden, so daß die Systemleistung beim Kopieren langer Zeichenfolgen möglicherweise sinkt.**

Beispiel:

```
SAY STORAGE()    → ' 248400
altinhalt = STORAGE('0004 0000'x,'Die Antwort')
CALL STORAGE '0004 0000'x,,32,'+'
```

STRIP() STRIP(Zeichenfolge[,{'B' | 'L' | 'T'}][,Füllzeichen])

Wird keines der wahlfrei zu verwendenden Argumente angegeben, entfernt die Funktion sowohl voran- als auch nachgestellte Leerzeichen aus dem Argument "Zeichenfolge". Das zweite Argument gibt an, ob nachgestellte (T - trailing), vorangestellte (L - leading) oder beide Arten von Leerzeichen (B - both) entfernt werden sollen. Das wahlfrei anzugebende "Füllzeichen" bezeichnet das zu entfernende Zeichen. Beispiel:

```
SAY STRIP('  Wie bitte?  ')    → Wie bitte?
SAY STRIP('  Wie bitte?  ','L') → Wie bitte?
SAY STRIP('++123+++','B','+')   → 123
```

SUBSTR() SUBSTR(Zeichenfolge,Start[,Länge][,Füllzeichen])

Gibt eine Teilzeichenfolge des Arguments "Zeichenfolge" zurück, die an der angegebenen Startposition beginnt und die angegebene Länge hat. Die Startposition muß eine positive Zahl sein. Als Standardlänge gilt der komplette Rest der Zeichenfolge. Ist die Teilzeichenfolge kürzer als die angeforderte Länge, wird sie rechts mit Leerzeichen oder dem angegebenen Füllzeichen aufgefüllt. Beispiel:

```
SAY SUBSTR('123456',4,2)    → 45
SAY SUBSTR('meinname',5,6,'=')   → name==
```

SUBWORD() SUBWORD(Zeichenfolge,n[,Länge])

Gibt eine Teilzeichenfolge des Arguments "Zeichenfolge" zurück, die mit dem n-ten Wort beginnt und deren Wortanzahl im Argument "Länge" definiert ist. Als Standardlänge gilt die restliche Länge der Zeichenfolge. Die zurückgegebene Zeichenfolge hat in keinem Fall voran- oder nachgestellte Leerzeichen. Beispiel:

```
SAY SUBWORD('Nun ist es Zeit  ',2,2)  → ist es
```

SYMBOL() SYMBOL(Name)

Überprüft, ob das Argument "Name" ein gültiges ARexx-Symbol ist. Ist dies nicht der Fall, gibt die Funktion die Zeichenfolge BAD zurück. Ist das Symbol nicht initialisiert, lautet die zurückgegebene Zeichenfolge LIT. Ist dem Symbol ein Wert zugeordnet, wird VAR zurückgegeben. Beispiel:

```
SAY SYMBOL('J')    → VAR
SAY SYMBOL('x')    → LIT
SAY SYMBOL('++')    → BAD
```

TIME() TIME(Option)

Gibt die aktuelle Systemzeit zurück oder steuert die interne Laufzeitmessung. Gültige Optionsschlüsselwörter sind:

CIVIL	Aktuelle Zeit im 12-Stunden-Format (a.m./p.m.) Stunde/Minuten
ELAPSED	Seit dem Programmstart vergangene Zeit in Sekunden
HOURS	Aktuelle Zeit in Stunden seit Mitternacht
MINUTES	Aktuelle Zeit in Minuten seit Mitternacht
NORMAL	Aktuelle Zeit im 24-Stunden-Format (Stunden/Minuten/Sekunden)
RESET	Setzt die Laufzeitmessung zurück
SECONDS	Aktuelle Zeit in Sekunden seit Mitternacht

Wird keine Option angegeben, gibt die Funktion die aktuelle Systemzeit im Format HH:MM:SS zurück. Beispiel:

```
/*Angenommen, es ist 1:02 Uhr nachts . . .*/
SAY TIME('C')      → 1:02 AM
SAY TIME('HOURS')      → 1
SAY TIME('M')      → 62
SAY TIME('N')      → 01:02:54
SAY TIME('S')      → 3720
call TIME('R')     /*Laufzeitmessung zurücksetzen*/
SAY TIME('E')      → .020
SAY TIME()         → 01:02:00
```

TRACE() TRACE(Option)

Stellt den Ablaufverfolgungsmodus (siehe Kapitel 6.1) entsprechend dem Schlüsselwort "Option" ein. Dafür muß eine der zulässigen alphabetischen oder Präfixoptionen verwendet werden. Die Funktion TRACE() ändert den Ablaufverfolgungsmodus auch während einer interaktiven Ablaufverfolgung. In diesem Fall werden TRACE-Befehle im Quellprogramm ignoriert. Rückgabewert ist der vor dem Funktionsaufruf aktive Modus. Dies ermöglicht die spätere Wiederherstellung des vorherigen Ablaufverfolgungsmodus. Beispiel:

```
/*Angenommen, der Ablaufverfolgungsmodus ist ?ALL*/
SAY TRACE('Results')    → ?A
```

TRANSLATE() TRANSLATE(Zeichenfolge[,Ausg.][,Eing.][,Füllz.])

Diese Funktion erstellt eine Umsetztabelle und ersetzt auf der Basis dieser Tabelle ausgewählte Zeichen aus dem Argument "Zeichenfolge". Wird nur das Argument "Zeichenfolge" angegeben, so wird es in Großbuchstaben umgesetzt. Bei Angabe einer Eingabetabelle wird die Umsetztabelle verändert, so daß Zeichen im Argument "Zeichenfolge", die in der Eingabetabelle vorkommen, durch die entsprechenden Zeichen in der Ausgabetabelle ersetzt werden. Zeichen, die über das Ende der Ausgabetabelle hinausgehen, werden durch das angegebene Füllzeichen oder durch Leerzeichen ersetzt. Die resultierende Zeichenfolge ist stets genau so lang wie die ursprüngliche Zeichenfolge. Die Ein- und Ausgabetabellen können beliebig lang sein. Beispiel:

```
SAY TRANSLATE("abcde","123","cbade","+") → 321++
SAY TRANSLATE("klein")    → KLEIN
SAY TRANSLATE("0110","10","01")    → 1001
```

TRIM() TRIM(Zeichenfolge)

Entfernt nachgestellte Leerzeichen aus dem Argument "Zeichenfolge". Beispiel:

```
SAY length(TRIM(' abc    ')) → 4
```

TRUNC() TRUNC(Zahl[,Stellen])

Gibt den ganzzahligen Teil des Arguments "Zahl", gefolgt von der angegebenen Anzahl Dezimalstellen zurück. Standardwert für die Dezimalstellen ist 0. Die Zahl wird bei Bedarf mit Nullen aufgefüllt. Beispiel:

```
SAY TRUNC(123.456)    → 123
SAY TRUNC(123.456,4)    → 123.4560
```

UPPER() UPPER(Zeichenfolge)

Setzt die Zeichenfolge in Großbuchstaben um. Diese Funktion hat den gleichen Effekt wie die Funktion TRANSLATE(Zeichenfolge), nur ist sie bei kurzen Zeichenfolgen etwas schneller. Beispiel:

```
SAY UPPER('Ein schöner Tag')    →  EIN SCHÖNER TAG
```

VALUE() VALUE(Name)

Gibt den Wert des Symbols zurück, das im Argument "Name" dargestellt ist. Beispiel:

```
/*Angenommen, J hat den Wert 12*/
SAY VALUE('j')    → 12
```

VERIFY() VERIFY(Zeichenfolge,Liste[,'MATCH'])

Gibt die Indexposition des ersten Zeichens im Argument "Zeichenfolge" zurück, das nicht im Argument "Liste" enthalten ist, oder 0, wenn alle Zeichen in der Liste stehen. Wird das Schlüsselwort MATCH angegeben, gibt die Funktion die Indexnummer des ersten Zeichens im Argument "Zeichenfolge" zurück, das im Argument "Liste" enthalten ist, oder 0, wenn keines der Zeichen in der Liste steht. Beispiel:

```
SAY VERIFY('123456','0123456789')    → 0
SAY VERIFY('123a56','0123456789')    → 4
SAY VERIFY('123a45','abcdefghij','m')    → 4
```

WORD() WORD(Zeichenfolge,n)

Gibt das n-te Wort aus dem Argument "Zeichenfolge" zurück oder eine leere Zeichenfolge, wenn die Zeichenfolge weniger als n Wörter hat. Beispiel:

```
SAY WORD('Nun ist es Zeit  ',2)   → ist
```

WORDINDEX() WORDINDEX(Zeichenfolge,n)

Gibt die Startposition des n-ten Worts im Argument "Zeichenfolge" zurück, oder 0, wenn die Zeichenfolge weniger als n Wörter hat. Beispiel:

```
SAY WORDINDEX('Nun ist es Zeit  ',3)   → 9
```

WORDLENGTH() WORDLENGTH(Zeichenfolge,n)

Gibt die Länge des n-ten Worts im Argument "Zeichenfolge" zurück. Beispiel:

```
SAY WORDLENGTH('vier fünf sechs',3)   → 5
```

WORDS() WORDS(Zeichenfolge)

Gibt die Anzahl der Wörter im Argument "Zeichenfolge" zurück. Beispiel:

```
SAY WORDS("Wie geht''s Dir?")   → 3
```

WRITECH() WRITECH(Datei,Zeichenfolge)

Schreibt das Argument "Zeichenfolge" in die angegebene logische Datei. Der Rückgabewert ist die Anzahl der geschriebenen Zeichen. Beispiel:

```
SAY WRITECH('Ausgabe','Test')   → 4
```

WRITELN() WRITELN(Datei,Zeichenfolge)

Schreibt das Argument "Zeichenfolge" in die angegebene logische Datei und fügt ein Zeichen für Zeilenvorschub an. Der Rückgabewert ist die Anzahl der geschriebenen Zeichen. Beispiel:

```
SAY WRITELN('Ausgabe','Test')    → 5
```

X2C() X2C(Zeichenfolge)

Wandelt eine aus Hexadezimalziffern bestehende Zeichenfolge in die (gepackte) Zeichendarstellung um. Leerzeichen sind im Argument "Zeichenfolge" an Bytegrenzen zulässig. Beispiel:

```
SAY X2C('12ab')    →'12ab'x
SAY X2C('12 ab')    → '12ab'x
SAY X2C(61)    → a
```

X2D() X2D(Hex-Zeichenfolge)

Wandelt eine Hexadezimalzahl in eine Dezimalzahl um. Beispiel:

```
SAY X2D('1f')    → 31
```

XRANGE() XRANGE([Start][,Ende])

Generiert eine Zeichenfolge, die aus allen Zeichen besteht, deren Codes numerisch zwischen den angegebenen Start- und Endzeichen liegen. Standardzeichen sind '00'x für Start und 'FF'x für Ende. Nur die jeweils ersten Zeichen der Argumente "Start" und "Ende" sind von Bedeutung. Beispiel:

```
SAY XRANGE()    → '00010203  . . .  FDFEFF'x
SAY XRANGE('a','f')    → 'abcdef'
SAY XRANGE(,'0A'x)→ '000102030405060708090A'x
```

5.5.3 Programmbeispiel

Das folgende Beispielprogramm illustriert viele der integrierten Funktionen, mit denen Zeichenfolgen manipuliert werden.

Programm 13. Changestrings.rexx

```
/*Dieses ARexx-Programm zeigt die Wirkung
integrierter Funktionen, die Zeichenfolgen
manipulieren. Die Funktionen lassen sich in zwei
Gruppen einteilen: solche, die einzelne Zeichen
verändern, und solche, die ganze Zeichenfolgen
verändern.*/

teststring1 = " every  good   boy   does   fine  "

/*Die erste Gruppe bilden die Funktionen STRIP(),
COMPRESS(), SPACE(), TRIM(), TRANSLATE(), DELSTR(),
DELWORD(), INSERT(), OVERLAY() und REVERSE().*/

/*STRIP() entfernt lediglich voran- und nachgestellte
Leerzeichen.*/

/*Drucken Sie zu Vergleichszwecken die
Originalzeichenfolge aus. Die Zeichenfolge wird mit
einem Punkt abgeschlossen, damit Sie sehen, was mit
den Leerzeichen am Ende der Zeichenfolge geschieht.*/
SAY " every  good   boy   does   fine  "

/*Die gleiche Zeichenfolge ohne voran- und nachgestellte
Leerzeichen*/
SAY STRIP(" every  good   boy   does   fine  ")"."

/*Gescheiterter Versuch, die voran- und nachgestellten
"e"'s zu entfernen*/
SAY STRIP(" every  good   boy   does   fine
",,"e")"."

/*Die "e"'s waren durch die noch immer vorhandenen,
voran- und nachgestellten Leerzeichen geschützt.
Durch Löschen dieser Leerzeichen werden die "e"'s den
Auswirkungen der Funktion STRIP() ausgesetzt*/
SAY STRIP("every  good   boy   does   fine",,"e")"."
```

```
/*Nun werden die "e"'s und die Leerzeichen aus der
Originalzeichenfolge gelöscht*/
SAY STRIP(" every good boy does fine ",,"
e")"."

/*Nun wird die weiter oben definierte Variable
"teststring1" verwendet. Entfernung nur der
nachgestellten Leerzeichen aus der
Testzeichenfolge.*/
SAY STRIP(teststring1, T)"."

/*Entfernung der nachgestellten Leerzeichen und des
"e"*/
SAY STRIP(teststring1,T," e")"."
.
/*Compress() entfernt Zeichen überall innerhalb einer
Zeichenfolge.  Hier werden alle Leerzeichen aus der
Testzeichenfolge teststring1 gelöscht*/
SAY COMPRESS(teststring1)

CALL TIME('r')
SAY TIME('Civil') /*Uhrzeit HH:MM{AM | PM}*/
SAY TIME('h')     /*Stunden seit Mitternacht*/
SAY TIME('m')     /*Minuten seit Mitternacht*/
SAY TIME('s')     /*Sekunden seit Mitternacht*/
SAY TIME('e')     /*Vergangene Zeit seit dem
Programmstart*/

/*Funktion: TRACE  Verwendung: TRACE([Option])*/
SAY TRACE()
SAY TRACE(TRACE())    /*Unverändert lassen*/
/*Funktion: TRANSLATE
Verwendung:
TRANSLATE(Zeich.folge[,Ausgabe][,Eingabe][,Füllz.])*/
SAY TRANSLATE('aBCdef')    /*Umwandeln in
Großbuchstaben*/
SAY TRANSLATE ('abcdef','1234')
SAY TRANSLATE('654321','abcdef','123456')
SAY TRANSLATE('abcdef','123','abcdef','+')

/*Funktion: TRIM  Verwendung: TRIM(Zeichenfolge)*/
SAY TRIM('   abc  ')

/*Funktion: TRUNC  Verwendung:
TRUNC(Zahl[,Stellen])*/
SAY TRUNC(123.456)
SAY TRUNC(134566.123,2) 'DM'
```

```
/*Funktion: UPPER  Verwendung: UPPER(Zeichenfolge)*/
SAY UPPER('aBCdef12äöÜ')

/*Funktion: VALUE  Verwendung: VALUE(Name)*/
abc = 'Mein Name'
SAY VALUE('abc')

/*Funktion: VERIFY  Verwendung:
VERIFY(Zeichenfolge,Liste[,'M'])*/
SAY VERIFY('123a45','0123456789')
SAY VERIFY('abc3de','012456789','M')

/*Funktion: WORD  Verwendung: WORD(Zeichenfolge,n)*/
SAY WORD('Nun ist es Zeit',3)

/*Funktion: WORDINDEX  Verwendung:
WORDINDEX(Zeichenfolge,n)*/
SAY WORDINDEX('Nun ist es Zeit  ',3)

/*Funktion: WORDLENGTH  Verwendung:
WORDLENGTH(Zeichenfolge,n)*/
SAY WORDLENGTH('Nun ist es Zeit  ',4)

/*Funktion: WORDS  Verwendung: WORDS(Zeichenfolge)*/
SAY WORDS('Nun ist es Zeit')

/*Funktion: WRITECH  Verwendung:
WRITECH(Datei,Zeichenfolge)*/
IF OPEN('test','ram:test$$','W') THEN DO
   SAY WRITECH('test','Hallo') /*Zeichenfolge
   schreiben*/
   CALL CLOSE 'test'
   END

/*Funktion: WRITELN  Verwendung:
WRITELN(Datei,Zeichenfolge)*/
IF OPEN('test','ram:test$$','W') THEN DO
   SAY WRITELN('test','Hallo')
   /*Zeichenfolge (plus Zeilenvorschubzeichen)
   schreiben*/
   CALL CLOSE 'test'
   END

/*Funktion: X2C  Verwendung: X2C(hexadezimale
Zeichenfolge)*/
SAY X2C('616263')   /*Umwandeln in Zeichen
(gepackt)*/
```

```
/*Funktion: XRANGE  Verwendung:
XRANGE([Start][,Ende])*/
SAY C2X(xrange('f0'x))
SAY XRANGE('a','g')
EXIT
```

Die Ausgabe des Testprogramms 13 sieht folgendermaßen aus:

```
 every  good   boy   does   fine
every  good   boy   does   fine.
 every  good   boy   does   fine .
very  good   boy   does   fin.
very  good   boy   does   fin.
 every  good   boy   does   fine.
 every  good   boy   does   fin.
everygoodboydoesfine
1:23PM   /*Diese Testergebnisse hängen von der
Uhrzeit ab, */
13       /*zu der Sie das Programm aufrufen.*/
803
48199
0.80
N
N
ABCDEF
abcdef
fedcba
123+++
   abc
123
134566.12 DM
ABCDEF12
Mein Name
4
0
es
8
4
4
7
8
abc
F0F1F2F3F4F5F6F7F8F9FAFBFCFDFEFF
abcdefg
```

5.6 Funktionen der Bibliothek REXXSupport.Library

Die in diesem Abschnitt aufgelisteten Funktionen sind Teil der Bibliothek REXXSupport.library. Sie können nur verwendet werden, wenn diese Bibliothek geöffnet ist. Das folgende Beispiel zeigt, wie Sie diese Bibliothek öffnen können.

Programm 14. OpenLibrary.rexx

```
/*Rexxsupport.library öffnen, falls noch nicht
geschehen.*/
IF ~ SHOW('L', "rexxsupport.library") THEN DO
/*Ist sie schon geöffnet? Wenn nicht, jetzt
versuchen, sie zu öffnen*/
IF ADDLIB('rexxsupport.library', 0, -30,0)
THEN SAY "Rexxsupport.library geöffnet."
ELSE DO
   SAY 'ARexx-Unterstützungsbibliothek nicht
verfügbar, Programm wird verlassen'
   EXIT 10     /*Bei Scheitern von ADDLIB() wird das
Programm verlassen*/
   END
END
```

ALLOCMEM() ALLOCMEM(Länge[,Attribut])

Reserviert einen Speicherblock der angegebenen Länge aus dem Pool des freien Systemspeichers und gibt dessen Adresse in Form einer 4-Byte-Zeichenfolge zurück. Der wahlfrei anzugebende Parameter "Attribut" muß eine standardmäßige EXEC-Kennung sein, die als 4-Byte-Zeichenfolge anzugeben ist. Standardattribut ist 1 für "PUBLIC" für allgemein zugänglichen (nicht vorher gelöschten) Speicher. Weitere Informationen über Speichertypen und Attributparameter finden sie in der Beschreibung von Bibliotheken in den *Amiga ROM Kernel Reference Manuals* (offizielle Systemdokumentation für Programmierer).

Diese Funktion sollten Sie immer dann verwenden, wenn Speicherplatz für externe Programme reserviert werden soll. Die Freigabe von nicht mehr benötigtem Speicherbereich liegt in der Verantwortung des Benutzers (siehe auch FREEMEM()). Beispiel:

```
SAY C2X(ALLOCMEM(1000))    → 00050000
SAY C2X(ALLOCMEM (1000,'00 01 00 0 1'X )) → 00228400
/*1000 Byte geCLEARter (auf 0 gesetzter) allgemein
zugänglicher Speicherplatz*/
```

CLOSEPORT() CLOSEPORT(Name)

Schließt den Message-Port, der im Argument "Name" angegeben wurde. Dieser muß zuvor innerhalb des aktuellen ARexx-Programms mit dem Aufruf OPENPORT() zugeordnet worden sein. Alle empfangenen Nachrichten, auf die noch nicht geantwortet (REPLY) wurde, werden automatisch mit dem Rückgabewert 10 zurückgegeben (siehe auch OPENPORT()). Beispiel:

```
CALL CLOSEPORT meinport
```

FREEMEM() FREEMEM(Adresse,Länge)

Gibt einen Speicherblock der angegebenen Länge frei (d. h. an den Pool des freien Systemspeichers zurück). Das Argument "Adresse" ist eine 4-Byte-Zeichenfolge, die sich in der Regel aus einem vorherigen Aufruf der Funktion ALLOCMEM() ergibt. FREEMEM() kann nicht zur Freigabe von Speicherplatz verwendet werden, der mit der internen ARexx-Speicherreservierungsfunktion GETSPACE() reserviert wurde. Der Rückgabewert ist ein boolesches Erfolgskennzeichen (siehe auch ALLOCMEM()). Beispiel:

```
Speicheranforderung = 1024 /* Speichergröße*/
MeinSp = ALLOCMEM(Speicheranforderung)
SAY C2X(MeinSp)                    → 07C987B0
SAY FREEMEM(MeinSp, Speicheranforderung)    → 1
/*Oder: SAY FREEMEM('07C987B0'x,1024)*/
```

Achtung **Vor Abschluß des Programms muß mit der Funktion FREEMEM() der zuvor mit ALLOCMEM() zugeordnete Speicherplatz wieder freigegeben werden. Andernfalls riskieren Sie einen Systemabsturz. Zumindest stünde aber der betreffende Speicherplatz erst nach dem nächsten Systemstart (Reboot) wieder zur Verfügung.**

GETARG() GETARG(Paket[,n])

Extrahiert ein Kommando, einen Funktionsnamen oder eine Argumentzeichenfolge aus einem Nachrichtenpaket. Das Argument "Paket" muß eine 4-Byte-Adresse sein, die sich aus einem vorherigen Aufruf der Funktion GETPKT() ergeben hat. Das wahlfreie Argument n bezeichnet die Position, an der die zu extrahierende Zeichenfolge steht, wobei n kleiner oder gleich der Gesamtzahl der Argumente in diesem Paket sein muß. Befehle und Funktionsnamen befinden sich stets an Position 0. Funktionspakete können Argumentzeichenfolgen an den Positionen 1 bis 15 aufweisen. Beispiel:

```
Kommando = GETARG(Paket)
Funktion = GETARG(Paket,0)   /*Name Zeichenfolge*/
arg1 = GETARG(Paket,1)       /*Erstes Argument*/
```

GETPKT() GETPKT(Name)

Überprüft, ob bei dem im Argument "Name" angegebenen Message-Port Nachrichten vorliegen. Der angegebene Message-Port muß zuvor innerhalb des aktuellen ARexx-Programms mit dem Aufruf OPENPORT() geöffnet worden sein. Der zurückgegebene Wert ist die 4-Byte-Adresse des ersten Nachrichtenpakets oder '0000 0000'x, wenn keine Pakete vorlagen.

Die Funktion meldet sofort, ob ein Paket in der Warteschlange des Message-Ports steht oder nicht. Programme sollten niemals so gestaltet sein, daß sie in einer engen Schleife permanent diese Abfrage aufrufen. Wenn das Programm bis zum Eintreffen des nächsten Nachrichtenpakets keine Aktionen auszuführen hat, sollte es die Funktion WAITPKT() aufrufen und damit die Weiterbearbeitung anderer Tasks ermöglichen (siehe auch WAITPKT()). Beispiel:

```
packet = GETPKT('MeinPort')
```

OPENPORT() OPENPORT(Name)

Erstellt einen allgemein zugänglichen Message-Port mit dem angegebenen Namen. Der zurückgegebene boolesche Wert gibt Aufschluß darüber, ob der Port erfolgreich geöffnet wurde. Wenn bereits ein Port mit diesem Namen existiert oder ein Signalbit nicht reserviert werden konnte, kommt es zu einem Initialisierungfehler. Der Message-Port wird als "Port-Ressourcen-Knoten" angelegt und in die globale Datenstruktur des Programms integriert. Ports werden automatisch geschlossen, wenn das Programm verlassen wird, und alle noch nicht verarbeiteten Meldungen werden an die Absender zurückgeschickt (siehe auch CLOSEPORT()). Beispiel:

```
erfolg = OPENPORT("MeinPort")
```

REPLY() REPLY(Paket,rc)

Schickt ein Nachrichtenpaket an den Absender zurück, wobei das Primärergebnisfeld auf den im Argument "rc" (Return Code) angegebenen Wert gesetzt wird. Das Sekundärresultat wird gelöscht. Das Argument "Paket" ist in Form einer 4-Byte-Adresse anzugeben, für "RC" ist eine ganze Zahl anzugeben. Beispiel:

```
CALL REPLY(Paket,10)  /*Fehlerrückgabe*/
```

SHOWDIR() SHOWDIR(Verzeichnis[,'ALL' | 'FILE' | 'DIR'][,Füllz.])

Gibt den Inhalt des angegebenen Verzeichnisses als eine Folge von jeweils durch Leerzeichen voneinander getrennten Namen zurück. Der zweite Parameter ist ein Optionsschlüsselwort, über das gesteuert wird, ob alle Einträge (ALL), nur Dateien (FILE) oder nur Unterverzeichnisse (DIR) des betreffenden Verzeichnisses berücksichtigt werden sollen. Beispiel:

```
SAY SHOWDIR('SYS:REXXC', 'f', ';')
→ WaitForPort;TS;TE;TCO;RXSET;RXLIB;RXC;RX;HI
```

SHOWLIST() SHOWLIST({'A' | 'D' | 'H' | 'I' | 'L' | 'M' | 'P' | 'R' | 'S' | 'T' | 'V' | 'W'}[,Name][,Füllzeichen])

Eine Liste wird durch Angabe ihres Anfangsbuchstabens ausgewählt. Mögliche Argumente sind:

A	(ASSIGNS and Assigned Device) Zuordnungen und zugeordnete Geräte
D	(Device Drivers) Gerätetreiber
H	(Handlers) Ein-/Ausgabeprogramme
I	(Interrupts) Unterbrechungen
L	(Libraries) Bibliotheken
M	(Memory List Items) Einträge der Speicherliste
P	(Ports)
R	(Resources) Ressourcen oder Betriebsmittel
S	(Semaphores) Semaphoren
T	(Tasks (Ready)) Bereite Tasks
V	(Volume Names) Datenträgernamen
W	(Waiting Tasks) Anstehende Tasks

Wird nur ein Argument angegeben, gibt SHOWLIST() eine Zeichenfolge mit Leerzeichen als Trennzeichen zurück. Wird ein Füllzeichen angegeben, wird dieses anstelle von Leerzeichen als Trennzeichen verwendet. Wird das Argument "Name" angegeben, gibt SHOWLIST() einen booleschen Wert zurück, der angibt, ob die angegebene Liste diesen Namen enthält. Beachten Sie bitte, daß bei Namen zwischen Groß- und Kleinschreibung unterschieden wird.

Während des Durchsuchens der Liste werden Task-Wechsel unterbunden, um eine exakte Momentaufnahme der aktuellen Liste zu erhalten.

```
SAY SHOWLIST('P')     → REXX MyCon
SAY SHOWLIST('P',,';')     → REXX;MyCon
SAY SHOWLIST('P','REXX')     → 1
```

STATEF() STATEF(Dateiname)

Gibt eine Zeichenfolge mit Informationen über eine externe Datei zurück. Die Zeichenfolge hat folgendes Format:

```
"{DIR | FILE} Länge Blöcke Schutz Tage Minuten Ticks
Kommentar."
```

"Länge" bezeichnet die Länge der Datei in Bytes, "Block" die Länge der Datei in Blöcken. Beispiel:

```
SAY STATEF("LIBS:REXXSupport.library")
/*mögliches Ergebnis "FILE 2524 5 --- RWED 4866 817
2088*/
```

WAITPKT() WAITPKT(Name)

Wartet an dem mit "Name" bezeichneten Message-Port auf den Empfang einer Nachricht. Dieser Port muß zuvor innerhalb des aktuellen ARexx-Programms durch den Aufruf der Funktion OPENPORT() geöffnet worden sein. Der boolesche Rückgabewert besagt, ob ein Nachrichtenpaket am Port vorliegt. Normalerweise ist der Rückgabewert 1 (Wahr), da die Funktion so lange wartet, bis sich am betreffenden Message-Port etwas tut.

Das Paket muß anschließend mit einem Aufruf der Funktion GETPKT() eingelesen und dann mit der Funktion REPLY() wieder zurückgeschickt werden. Alle eingegangenen, aber noch nicht zurückgeschickten Nachrichtenpakete werden beim Verlassen des ARexx-Programms automatisch mit dem Rückgabewert 10 beantwortet (Funktion REPLY). Beispiel:

```
CALL WAITPKT 'MeinPort'    /*Solange warten*/
```

Kapitel 6
Fehlersuche

ARexx bietet Funktionen zur Fehlersuche (engl. debugging) und Ablaufverfolgung (engl. tracing) auf Quelltextebene. Die Ablaufverfolgung zeigt während der Ausführung eines Programms ausgewählte Anweisungen innerhalb dieses Programms an. Trifft der Ablaufverfolgungsprozeß auf eine Klausel, werden ihre Zeilennummer, ihr Quelltext und die Klausel betreffende Informationen an der Konsole angezeigt.

Das interne Interrupt-System gibt dem ARexx-Programm die Möglichkeit, bestimmte synchrone oder asynchrone Ereignisse zu erkennen und entsprechend auf solche Ereignisse zu reagieren. Ereignisse wie Syntaxfehler oder externe Unterbrechungsanforderungen, die normalerweise zum Verlassen des Programms führen würden, können auf diese Weise abgefangen und durch geeignete Maßnahmen korrigiert werden.

6.1 Ablaufverfolgung

Die Ablaufverfolgungsoption bestimmt, welche Klauseln des Quelltexts verfolgt werden sollen. Sie besitzt zwei Modifizierungskennzeichen zur Steuerung der Kommandosperre und der Ablaufverfolgung im Dialogbetrieb (interaktiv). Ablaufverfolgungsoptionen können auf einen einzigen Buchstaben abgekürzt angegeben werden. Mögliche Optionen sind:

ALL	Alle Klauseln fallen unter die Ablaufverfolgung.
BACKGROUND	Es findet keine Ablaufverfolgung statt, und das Programm kann nicht zu einer interaktiven Ablaufverfolgung veranlaßt werden.
COMMANDS	Alle Kommandoklauseln fallen unter die Ablaufverfolgung, bevor sie an den externen Host gesendet werden. Von Null abweichende Rückgabewerte werden an der Konsole angezeigt.
ERRORS	Kommandos, die einen von Null abweichenden Rückgabewert generieren, werden nach Ausführung der Klausel verfolgt.
INTERMEDIATES	Alle Klauseln fallen unter die Ablaufverfolgung. Während der Auswertung der Ausdrücke werden Zwischenergebnisse angezeigt. Zu diesen Ergebnisssen gehören die aus Variablen abgerufenen Werte, erweiterte zusammenge-setzte Namen und die Ergebnisse von Funktionsaufrufen.
LABELS	Alle Sprungmarkenklauseln werden während ihrer Ausführung verfolgt. Eine Sprungmarke wird immer dann angezeigt, wenn die Steuerung an eine andere Stelle übertragen wird.
NORMAL (Standardwert)	Kommandoklauseln, deren Rückgabewerte die aktuellen Fehlergrenzwerte überschreiten, werden nach ihrer Ausführung in die Ablaufverfolgungs-liste aufgenommen, und eine Fehlermeldung wird angezeigt.
OFF	Die Ablaufverfolgung ist ausgeschaltet.
RESULTS	Alle Klauseln werden vor ihrer Ausführung in die Ablaufverfolgungsliste geschrieben, und die Ergebnisse jedes einzelnen Ausdrucks werden angezeigt. Werte, die Variablen durch einen der Befehle ARG, PARSE oder PULL zugeordnet werden, werden ebenfalls angezeigt. Diese Option empfiehlt sich, wenn Sie Programme einem allgemeinen Test unterziehen wollen.
SCAN	Dies ist eine spezielle Option, die alle Klauseln in die Ablaufverfolgungsliste schreibt und auf Fehler untersucht, aber die Ausführung der Anweisungen verhindert. Sie eignet sich besonders zur erst-maligen Kontrolle neu geschriebener Programme.

Der Ablaufverfolgungsmodus kann über den Befehl TRACE oder die integrierte Funktion TRACE() gesetzt werden. Die Ablaufverfolgung kann innerhalb eines Programms selektiv inaktiviert werden, um bereits getestete Komponenten von Programmen zu überspringen.

Jede auf der Konsole angezeigte Zeile der Ablaufverfolgung ist eingerückt, um die tatsächliche Steuerungs- bzw. Verschachtelungsebene der jeweiligen Klausel zu verdeutlichen. Ferner ist jede Zeile durch einen speziellen, dreistelligen Code gekennzeichnet (siehe dazu die folgende Tabelle 6-1). Dem Quelltext jeder Klausel wird die Zeilennummer innerhalb des Programms vorangestellt. Resultate und Zwischenresultate von Ausdrücken stehen in doppelten Anführungszeichen, so daß auch voran- und nachgestellte Leerzeichen sichtbar werden.

Tabelle 6-1. Spezielle dreistellige Codes

Code	Angezeigte Werte
+++	Kommando- oder Syntaxfehler
>C>	Erweiterter zusammengesetzter Name
>F>	Ergebnis eines Funktionsaufrufs
>L>	Sprungmarkenklausel
>O>	Ergebnis einer dyadischen Operation
>P>	Ergebnis einer Präfixoperation
>U>	Nicht initialisierte Variable
>V>	Wert einer Variablen
>>>	Ergebnis eines Ausdrucks oder einer Schablone
>.>	Wert eines "Platzhalter"-Tokens

6.1.1 Ausgabedaten der Ablaufverfolgung

Die Ausgabedaten der Ablaufverfolgung eines Programms werden stets in einen von zwei logischen Datenströmen eingespeist. Der Interpreter sucht zunächst nach einem Datenstrom mit dem Namen STDERR und leitet die Ausgabe dorthin, falls dieser Datenstrom existiert. Andernfalls geht die Ausgabe an den standardmäßigen

Ausgabedatenstrom STDOUT und wird mit der normalen Konsolenausgabe des Programms vermischt. Die Datenströme STDERR und STDOUT können vom Programm aus geöffnet und geschlossen werden, so daß der Programmierer das Ziel der Ablaufverfolgungsausgabe stets unter Kontrolle hat.

In manchen Fällen ist der Ausgabedatenstrom eines Programms nicht vorab definiert. Wird z. B. ein Programm von einer Host-Anwendung aufgerufen, die keine Ein- und Ausgabedatenströme bereitstellt, verfügt das Programm über keine Ausgabekonsole. Um auch für solche Programme eine Möglichkeit der Ablaufverfolgung zu schaffen, kann der residente Prozeß eine spezielle, globale Ablaufverfolgungskonsole öffnen, die von jedem aktiven Programm geöffnet werden kann. Wird diese Konsole geöffnet, öffnet der Interpreter automatisch einen Datenstrom mit dem Namen STDERR für jedes ARexx-Programm, in dem STDERR zur Zeit nicht definiert ist. Das Programm leitet dann seine Ablaufverfolgungsausgabe an diesen neuen Datenstrom um.

Mit dem Kommandohilfsprogramm TCO (TraCe Open) kann eine Konsole für globale Ablaufverfolgung geöffnet werden. ARexx-Programme leiten die Ausgabedaten ihrer Ablaufverfolgung automatisch in das neue Fenster um, das als standardmäßige AmigaDOS-Konsole geöffnet wird. Der Benutzer kann Position und Größe dieses Fensters beliebig verändern.

Die Ablaufverfolgungskonsole dient auch als Eingabedatenstrom für Programme während der interaktiven Ablaufverfolgung. Wenn ein Programm auf Eingabedaten zur Ablaufverfolgung wartet, müssen diese Daten über die Ablaufverfolgungskonsole (in deren Fenster) eingegeben werden. Beliebig viele laufende Programme können sich die Ablaufverfolgungskonsole teilen. Es empfiehlt sich allerdings, die Ablaufverfolgung immer nur für jeweils ein Programm zu aktivieren.

Die globale Konsole kann mit dem Kommando TCC (TraCe Close) geschlossen werden. Das Schließen wird allerdings erst ausgeführt, wenn alle Leseanforderungen an die Konsole bearbeitet wurden. Die Konsole wird also erst geschlossen, wenn alle aktiven Programme melden, daß sie die Konsole nicht mehr benutzen.

6.1.2 Kommandosperre

ARexx verfügt über einen Ablaufverfolgungsmodus, der Host-Kommandos unterdrückt, die sog. Kommandosperre. In diesem Modus werden Kommandoklauseln in der üblichen Weise ausgewertet, das Kommando wird jedoch nicht wirklich an den externen Host gesendet. Der Rückgabewert (RC) wird auf Null gesetzt. Auf diese Weise können Programme gestestet werden, die Kommandos mit einer potentiell zerstörenden Wirkung ausgeben, z. B. das Löschen von Dateien oder das Formatieren von Festplatten oder Disketten. Die Kommandosperre gilt jedoch nicht für Kommandoklauseln, die im Dialogbetrieb (interaktiv) eingegeben werden. Solche Kommandos werden stets ausgeführt, der Wert der Sondervariablen RC bleibt allerdings unverändert.

Die Kommandosperre kann in Verbindung mit jeder Ablaufverfolgungsoption eingesetzt werden. Sie wird durch das Zeichen "!" gesteuert, das allein stehen oder einer der alphabetischen Optionen in einem TRACE-Befehl vorangestellt werden kann. Mit jedem vorkommenden "!" wird vom zur Zeit aktiven Sperrmodus in den anderen Modus umgeschaltet. Die Kommandosperre wird aufgehoben, wenn die Ablaufverfolgung auf OFF gesetzt wird.

6.1.3 Interaktive Ablaufverfolgung

Die interaktive Ablaufverfolgung ist eine Fehlersuchhilfe, die es dem Benutzer ermöglicht, während der Ausführung eines Programms Quellanweisungen einzugeben. Über solche Anweisungen können Variablenwerte überprüft oder verändert, Kommandos ausgegeben oder auf andere Weise mit dem Programm in einen Dialog getreten werden. Jede gültige Sprachanweisung kann interaktiv eingegeben werden. Es gelten die gleichen Regeln und Einschränkungen wie für den Befehl INTERPRET. Insbesondere ist darauf zu achten, daß zusammengesetzte Anweisungen, wie z. B. DO und SELECT, innerhalb der eingegebenen Zeile vollständig sind.

Die interaktive Ablaufverfolgung kann mit jeder Ablaufverfolgungsoption verwendet werden. Im Dialogmodus legt der Interpreter nach jeder in die Ablaufverfolgungsliste geschriebenen Klausel eine

Pause ein und fordert mit dem Code "+++" zu einer Benutzereingabe auf. Bei jeder dieser Pausen hat der Benutzer drei verschiedene Antworten zur Auswahl:

- Eingabe einer Nullzeile (einfach Eingabetaste) - das Programm wird bis zum nächsten Pausenpunkt fortgesetzt.
- Eingabe eines Gleichheitszeichens (=) - die vorherige Klausel wird nochmals ausgeführt.
- Jede andere Eingabe wird als Fehlersuchanweisung behandelt; sie wird geprüft und ausgeführt.

Da der Interpreter nach erfolgter Ablaufverfolgung einer Klausel pausiert, werden die Pausenpunkte von der zur Zeit aktivierten Ablaufverfolgungsoption bestimmt. Die Ausführung mancher Befehle kann allerdings nicht ohne Risiko wiederholt werden, so daß der Interpreter nach Ausführung solcher Befehle nicht pausiert. Diese Befehle sind CALL, DO, ELSE, IF, THEN und OTHERWISE. Der Interpreter pausiert gleichfalls nicht nach Klauseln, die einen Ausführungsfehler generiert haben.

Die interaktive Ablaufverfolgung wird durch das Zeichen "?" gesteuert, das allein stehen oder mit einer der alphabetischen Optionen in einem TRACE-Befehl angegeben werden kann. Einer Option können beliebig viele "?" vorangestellt werden. Mit jedem vorkommenden "?" wird vom zur Zeit aktiven Modus in den anderen Modus umgeschaltet. Wäre z. B. die aktuelle Ablaufverfolgungsoption NORMAL, so würde mit "TRACE ?R" die Option RESULTS gesetzt und der interaktive Ablaufverfolgungsmodus aktiviert. Ein späteres "TRACE ?" würde die interaktive Ablaufverfolgung wieder ausschalten.

6.1.3.1 Fehlerbehandlung

Der ARexx-Interpreter stellt während der Fehlersuche eine spezielle Fehlerbehandlung zur Verfügung. Während der interaktiven Fehlersuche gefundene Fehler werden gemeldet, führen aber nicht zur Beendigung des Programmlaufs. Diese spezielle Verarbeitung gilt allerdings nur für im Dialogbetrieb eingegebene Anweisungen.

ARexx inaktiviert auch die internen Interrupt-Kennzeichen während der interaktiven Fehlersuche. Damit wird eine unbeab-

sichtigte Steuerungsübertragung verhindert, die durch einen Fehler oder das Antreffen einer nicht initialisierten Variable ausgelöst werden könnte. Wird jedoch der Befehl "SIGNAL Sprungmarke" eingegeben, findet der Steuerungstransfer statt, und eventuell verbleibende interaktive Eingabedaten bleiben unberücksichtigt. Der Befehl SIGNAL kann auch weiterhin dazu verwendet werden, die Interrupt-Kennzeichen zu ändern. Die neuen Einstellungen treten in Kraft, wenn der Interpreter in den normalen Verarbeitungsmodus zurückkehrt.

Jeder ARexx-Task initialisiert seinen Fehlergrenzwert entsprechend dem Grenzwert des Klienten (normalerweise 10). Damit wird das Ausdrucken unnötiger Kommandofehler unterbunden. Der Fehlergrenzwert kann mit OPTIONS FAILAT verändert werden. Kommandofehler (RC > 0) und fehlgeschlagene Kommandos (RC >= FAILAT) können mit SIGNAL ON ERROR und SIGNAL ON FAILURE separat verfolgt werden.

6.1.3.2 Externes Ablaufverfolgungskennzeichen

ARexx verfügt über ein externes Ablaufverfolgungskennzeichen, mit dem Programme in den interaktiven Ablaufverfolgungsmodus gezwungen werden können. Wenn dieses Kennzeichen gesetzt wird (mit dem Kommandohilfsprogramm TS (Trace Set)), geht jedes Programm in den interaktiven Ablaufverfolgungsmodus über (sofern es sich nicht bereits in diesem Modus befindet). Die interne Ablaufverfolgungsoption wird auf RESULTS gesetzt, es sei denn, sie ist gegenwärtig auf INTERMEDIATES oder SCAN gesetzt. In diesen Fällen bleibt sie unverändert. Programme, die aufgerufen werden, während das externe Ablaufverfolgungskennzeichen gesetzt ist, beginnen die Ausführung im interaktiven Ablaufverfolgungsmodus.

Das externe Ablaufverfolgungskennzeichen bietet eine Möglichkeit, die Kontrolle über Programme wieder zu gewinnen, die sich in Schleifen befinden oder nicht reagieren. Sobald ein Programm in den interaktiven Ablaufverfolgungsmodus übergeht, kann der Benutzer die Programmanweisungen Schritt für Schritt durchgehen und das Problem diagnostizieren. Die externe Ablaufverfolgung ist ein globales Kennzeichen, d. h. sie wirkt sich auf alle gegenwärtig aktiven Prozesse aus. Das Ablaufverfolgungskennzeichen bleibt solange gesetzt, bis es mit dem Kommandohilfsprogramm TE (Trace

End) gelöscht wird. Jedes Programm verwaltet eine interne Kopie des letzten Status des Ablaufverfolgungskennzeichens und setzt seine eigene Ablaufverfolgungsoption auf OFF, wenn es feststellt, daß das Ablaufverfolgungskennzeichen gelöscht wurde. Programme im Ablaufverfolgungsmodus BACKGROUND reagieren nicht auf das externe Ablaufverfolgungskennzeichen.

6.2 Interrupts

ARexx unterhält ein internes Interrupt-System zum Erkennen und Abfangen bestimmter Fehlerbedingungen. Wird ein Interrupt aktiviert und seine entsprechende Bedingung tritt ein, kommt es zur Übertragung der Steuerung an die dieser Unterbrechung zugehörige Sprungmarke. Auf diese Weise kann ein Programm die Steuerung behalten, obwohl die vorliegenden Bedingungen normalerweise zum Abbruch des Programms führen würden. Die Interrupt-Bedingungen können durch synchrone Ereignisse (z. B. Syntaxfehler) oder asynchrone Ereignisse z. B. Unterbrechungsanforderungen über die Tastatureingabe Ctrl-C) ausgelöst werden.

Hinweis Solche internen Interrupts haben nichts mit dem Hardware-Interrupt-System zu tun, das vom Betriebssystem EXEC verwaltet wird.

Der jedem Interrupt zugeordnete Name ist jeweils die Sprungmarke, an die die Steuerung übergeben wird. Damit überträgt ein Interrupt SYNTAX die Steuerung also an die Sprungmarke "SYNTAX:". Interrupts können mit dem Befehl SIGNAL aktiviert oder inaktiviert werden. Der Befehl "SIGNAL ON SYNTAX" aktiviert z. B. den Interrupt SYNTAX.

Die von ARexx unterstützten Interrupts sind:

BREAK_C Fängt eine von AmigaDOS generierte Unterbrechungsanforderung Ctrl-C ab (d. h. sie wird entdeckt und als Signal interpretiert, aber nicht als normale Eingabe). Ist der Interrupt nicht aktiviert, bricht das Programm sofort ab, gibt die Fehlermeldung "Execution halted" (Ausführung abgebrochen) aus und gibt den Fehlercode 2 zurück.

BREAK_D Fängt eine von AmigaDOS generierte Unterbrechungsanforderung Ctrl-D ab. Ist der Interrupt nicht aktiviert, wird die Unterbrechungsanforderung ignoriert.

BREAK_E Fängt eine von AmigaDOS generierte Unterbrechungsanforderung Ctrl-E ab. Ist der Interrupt nicht aktiviert, wird die Unterbrechungsanforderung ignoriert.

BREAK_F Fängt eine von AmigaDOS generierte Unterbrechungsanforderung Ctrl-F ab. Ist der Interrupt nicht aktiviert, wird die Unterbrechungsanforderung ignoriert.

ERROR Dieser Interrupt wird von jedem Host-Kommando generiert, das einen von Null abweichenden Fehlercode zurückgibt.

HALT Ist dieser Interrupt aktiviert, wird eine externe Unterbrechungsanforderung abgefangen. Andernfalls bricht das Programm sofort ab und gibt die Fehlermeldung "Execution halted" (Ausführung abgebrochen) aus.

IOERR Ist dieser Interrupt aktiviert, werden vom E/A-System entdeckte Fehler abgefangen.

NOVALUE Ist diese Bedingung aktiviert, kommt es zu einem Interrupt, wenn eine nicht initialisierte Variable verwendet wird. Die Auslösung dieses Interrupts kann innerhalb eines Ausdrucks, im Befehl UPPER oder in der integrierten Funktion VALUE() erfolgt sein.

SYNTAX Dieser Interrupt generiert einen Syntax- oder Ausführungsfehler. Nicht alle Fehler dieser Art können abgefangen werden. Manche Fehler treten vor der Ausführung eines Programms auf. Diejenigen, die von der externen ARexx-Schnittstelle entdeckt werden, können vom Interrupt SYNTAX nicht abgefangen werden.

Wenn ein Interrupt zwangsläufig einen Steuerungstransfer ("Sprung") herbeiführt, werden alle zur Zeit aktiven Steuerungsbereiche inaktiviert. Der Interrupt, der die Steuerungsübertragung auslöste, wird inaktivert. Dies verhindert die Entstehung einer rekursiven Interrupt-Schleife. Betroffen sind allerdings nur die Steuerungsstrukturen der aktuellen Umgebung. Ein innerhalb einer Funktion generierter Interrupt hat also keine Auswirkungen auf die Umgebung der Aufrufebene.

Das Auftreten eines Interrupts wirkt sich auf zwei Sondervariablen aus:

SIGL Ist stets auf die aktuelle Zeilennummer gesetzt, bevor es zum Steuerungstransfer kommt. Damit kann festgestellt werden, welche Zeile des Quelltextes ausgeführt wurde.

RC Ist auf den Fehlercode gesetzt, der die Bedingung auslöste. Bei ERROR-Interrupts ist dieser Wert der Rückgabewert eines Kommandos und kann normalerweise als Einstufung des Fehlerschweregrades interpretiert werden. Der Wert des SYNTAX-Interrupts ist stets ein ARexx-Fehlercode.

Interrupts sind sehr nützlich bei der Fehlerbehebung. Dazu gehört die Information externer Programme über das Auftreten eines Fehlers oder das Melden weiterer Diagnosedaten zur Eingrenzung des Problems. Beispielprogramm 15 zeigt, wie bei Feststellung eines Syntaxfehlers ein Kommando "message" an den externen Host ausgegeben wird.

Programm 15. Interrupt.rexx

```
/*Ein Makro-Programm für 'MeinEdit'*/
SIGNAL ON SYNTAX        /*Interrupt aktivieren*/
(normale Verarbeitungsbefehle)
EXIT
SYNTAX:                 /*Syntaxfehler entdeckt*/
ADDRESS 'MeinEdit'
'message' 'error' RC errortext(RC)
EXIT 10
```

Kapitel 7

Syntaxanalyse

Bei der Syntaxanalyse (engl. Parsing) werden Teilzeichenfolgen aus einer Zeichenfolge extrahiert und Variablen zugeordnet. Die Syntaxanalyse wird mit dem Befehl PARSE oder seinen Varianten ARG und PULL aufgerufen. Die Eingabe für diese Operation heißt Analysezeichenfolge und stammt aus verschiedenen Quellen, z. B. aus Argumentzeichenfolgen, Ausdrücken oder auch von der Konsole.

Funktionen zur Manipulation von Zeichenfolgen, wie SUBSTR() und INDEX(), können zur Syntaxanalyse eingesetzt werden. Der Befehl PARSE ist allerdings weit effizienter, besonders wenn viele Felder aus einer Zeichenfolge extrahiert werden.

7.1 Schablonen

Die Syntaxanalyse wird von einer Schablone gesteuert. Unter einer Schablone versteht man eine Token-Gruppe, in der die Variablen angegeben sind, die mit Werten ausgestattet werden sollen, und die Art und Weise, wie die Wertzeichenfolgen zu extrahieren sind. Die Anordnung der Tokens in der Schablone bestimmt, ob das Token eines der beiden Basisobjekte einer Schablone ist: eine Marke oder ein Ziel.

Marke Legt die Start- und Endposition in der Analysezeichenfolge oder eine Suchposition fest.

Ziel Ein Symbol, dem durch die Syntaxanalyseoperation ein Wert zugeordnet wird. Dieser Wert ist die durch die Position der Marken definierte Teilzeichenfolge.

7.1.1 Marken

Es gibt drei Arten von Markenobjekten:

Absolute Marken	Tatsächliche Indexposition in der Analysezeichenfolge.
Relative Marken	Position relativ zur aktuellen Position (positiver oder negativer Wert).
Mustermarken	Vergleicht das Suchmuster mit der Analysezeichenfolge, beginnend an der aktuellen Suchposition.

7.1.2 Ziele

Ziele können ebenso wie Marken auf die Suchposition Einfluß haben, wenn Wertzeichenfolgen durch "Tokenisierung", also die Zerlegung von Zeichenfolgen in Tokens, extrahiert werden. Syntaxanalyse durch Tokenisierung extrahiert Wörter (Tokens) aus der Analysezeichenfolge und wird immer dann verwendet, wenn auf ein Ziel unmittelbar ein weiteres Ziel folgt. Während der Tokenisierung wandert die aktuelle Suchposition über eventuelle Leerzeichen hinweg zum Anfang des nächsten Wortes weiter. Der Endeindex ist die Position genau hinter dem Ende des Worts, so daß die Wertzeichenfolge weder voran- noch nachgestellte Leerzeichen aufweist.

Ziele werden durch variable Symbole angegeben. Der Platzhalter Punkt (.) ist ein spezieller Zieltyp und verhält sich genau wie ein normales Ziel, mit der Ausnahme, daß er nicht über einen zugeordneten Wert verfügt.

7.1.2.1 Schablonenobjekte

Jedes Objekt in einer Schablone wird durch ein oder mehrere Tokens angegeben:

Symbole Ein Symbol kann ein Ziel oder eine Marke angeben. Um eine Marke handelt es sich, wenn es auf einen Operator (+, - oder =) folgt und der Symbolwert als absolute oder relative Position verwendet wird. Symbole in runden Klammern sind dagegen Mustermarken, und der Symbolwert dient als Musterzeichen-folge. Symbole sind Ziele, wenn keiner der beschriebenen Fälle zutrifft und das Symbol eine Variable ist. Feste Symbole bezeichnen stets absolute Marken und müssen ganze Zahlen sein. Einzige Ausnahme bildet der Platzhalter (.) als Zielan-gabe.

Zeichen-folgen Eine Zeichenfolge steilt immer eine Mustermarke dar.

Runde Klammern Ein in runden Klammern stehendes Symbol ist eine Mustermarke. Der Wert des Symbols dient als Muster-zeichenfolge. Das Symbol kann fest oder variabel sein, in der Regel ist es aber eine Variable, da ein festes Muster einfacher als Zeichenfolge angegeben werden könnte.

Operatoren Die drei Operatoren (+, – und =) sind innerhalb einer Schablone gültig und müssen einem festen oder variablen Symbol vorangestellt sein. Der Wert des Symbols wird als Marke verwendet und muß ganzzahlig sein. Die Operatoren "+" und "–" bezeichnen eine relative Marke, wobei der Operator "–" einen negativen Wert definiert. Der Operator "=" bezeichnet eine absolute Marke und kann weggelassen werden, wenn die Marke durch ein festes Symbol definiert ist.

Kommas Das Komma (,) markiert das Ende einer Schablone. Es dient auch als Trennzeichen, wenn innerhalb eines Befehls mehrere Schablonen angegeben werden. Vor der Verarbeitung aufeinanderfolgender Schablonen erhält der Interpreter jeweils eine neue Analysezeichenfolge. Bei einigen Quellprogrammoptionen ist die neue Zeichenfolge mit der vorherigen identisch. Die Optionen ARG, EXTERNAL und PULL liefern in der Regel eine von der vorherigen abweichende Zeichenfolge, ebenso wie die Option VAR, wenn die Variable verändert wurde.

Das Kommandoanalyseprogramm der ARexx-Schnittstelle wurde dahingehend vereinheitlicht, daß Sequenzen mit doppelten Begrenzungszeichen innerhalb einer (in Anführungszeichen stehenden) Zeichenfolgendatei erkannt werden. Die Konvention für die Verwendung von Anführungszeichen ist besonders geeignet für kleinere Programme. In komplexeren Programmen kann es relativ leicht vorkommen, daß die Verschachtelungsebenen für das Setzen von Anführungszeichen nicht mehr ausreichen. Einzelne und doppelte Anführungszeichen sind innerhalb eines REXX-Progamms äquivalent, allerdings trifft die externe Umgebung möglicherweise eine Unterscheidung zwischen den verschiedenen Anführungszeichen.

AmigaDOS verwendet das doppelte Anführungszeichen. Zeichenfolgen, die von einer Shell aus eingegeben werden, müssen mit einem doppelten Anführungszeichen beginnen, besonders, wenn sie Semikolons enthalten sollen. Beispiel:

```
RX "SAY 'Das is'' ja ''n Ding - ich glaub'', mich
tritt ''n Pferd!' "
→ Das is' ja 'n Ding - ich glaub', mich tritt 'n
Pferd!

RX "SAY '""Hallo!""'"     → "Hallo!"
```

7.2 Suchvorgang

Suchpositionen werden in Form eines Index innerhalb der Analysezeichenfolge angegeben und liegen im Bereich von 1 (Anfang der Zeichenfolge) bis zur Länge der Zeichenfolge plus 1 (hinter dem Ende der Zeichenfolge).

Die in Form von zwei Suchindizes angegebene Teilzeichenfolge enthält die Zeichen von der Startposition bis eine Stelle vor der Endposition (die Endposition selbst gehört also nicht mehr dazu!). Die Indizes 1 und 10 schließen also nur die Positionen 1 bis 9 der Analysezeichenfolge ein. Ist der Wert des zweiten Suchindex kleiner oder gleich dem ersten, wird der Rest (beginnend mit der Startposition) der Analysezeichenfolge als Teilzeichenfolge verwendet. Die Schablonenangabe:

```
PARSE ARG 1 alle 1 erst zweit
```

ordnet also die gesamte Analysezeichenfolge der Variablen ALLE zu. Befindet sich der aktuelle Suchindex bereits am Ende der Analysezeichenfolge, verbleibt als Rest eine leere Zeichenfolge.

Wenn eine Mustermarke mit der Analysezeichenfolge abgeglichen wird, ist die Markenposition der Index des ersten Zeichens des gefundenen Musters oder aber das Ende der Zeichenfolge, wenn keine Entsprechung gefunden wurde. Immer wenn eine Entsprechung gefunden wird, wird das Muster aus der Zeichenfolge gelöscht. Dies ist der einzige Fall, bei dem die Analysezeichenfolge im Verlauf der Syntaxanalyse verändert wird.

Schablonen werden von links nach rechts abgesucht, wobei der Anfangs-Suchindex auf 1 gesetzt ist. Jedesmal, wenn ein Markenobjekt angetroffen wird, wird die Suchposition entsprechend dem Typ und dem Wert der Marke aktualisiert. Wird ein Zielobjekt gefunden, so wird der zugeordnete Wert durch Prüfung des nächsten Schablonenobjekts festgelegt. Ist das nächste Objekt ein weiteres Ziel, so wird die Wertzeichenfolge durch Tokenisierung der Analysezeichenfolge bestimmt. Andernfalls dient die aktuelle Suchposition als Anfang der Wertzeichenfolge und die durch die darauffolgende Marke definierte Position als Ende der Wertzeichenfolge.

Der Suchvorgang wird fortgesetzt, bis alle Objekte der Schablone verwendet worden sind. Jedem Ziel wird ein Wert zugeordnet. Wenn die Analysezeichenfolge vollständig ausgeschöpft wurde, wird jedem danach folgenden Ziel die Nullzeichenfolge zugeordnet.

7.3 Beispiele zur Syntaxanalyse

7.3.1 Syntaxanalyse durch Tokenisierung

Computerprogramme zerlegen eine Zeichenfolge häufig in ihre Bestandteile - Wörter oder Tokens. Dies erreicht man mittels einer Schablone, die sich vollständig aus Variablen (Zielen) zusammensetzt.

```
/*Angenommen es wurde "Hammer 1 Stück DM600.00"
eingegeben*/
PULL teil anz einh preis .
```

In diesem Beispiel wird die Eingabezeile aus dem Befehl PULL in Wörter zerlegt und den Variablen in der Schablone zugewiesen. Die Variable "teil" erhält den Wert "Hammer", anz wird auf "1" gesetzt, einh auf "Stück", und preis erhält den Wert "DM600.00". Der abschließende Platzhalter (.) erhält einen Nullwert, da die Eingabe nur aus vier Wörtern besteht. Allerdings führt er dazu, daß die vorherige Variable (preis) einen tokenisierten Wert erhält. Würde der Platzhalter weggelassen, so würde der gesamte Rest der Analysezeichenfolge der Variable "preis" zugewiesen, so daß dieser ein Leerzeichen vorangestellt würde.

```
antwort = "Nur Amiga macht es möglich."
DO forever
   PARSE VAR antwort erst antwort
/*Erstes Wort in 'erst' und den Rest in 'antwort'
stellen.*/
   IF erst ='' THEN LEAVE
   /*Beenden, wenn keine weiteren Wörter mehr
vorhanden sind*/
   SAY antwort
   END
```

Das erste Wort einer Zeichenfolge wird entfernt und der Rest wieder in die Zeichenfolge gestellt. Der Prozeß wird so oft wiederholt, bis keine Wörter mehr extrahiert werden können. Daraus ergibt sich folgende Ausgabe:

```
Amiga macht es möglich.
macht es möglich.
es möglich.
möglich.
```

7.3.2 Syntaxanalyse nach Mustern

Mustermarken extrahieren die gewünschten Felder. Das "Muster" ist in diesem Fall sehr einfach — es besteht nur aus einem einzigen Zeichen. Es könnte aber auch eine beliebig lange Zeichenfolge gewählt werden. Diese Form der Syntaxanalyse empfiehlt sich besonders, wenn in der Analysezeichenfolge Begrenzungszeichen vorkommen.

```
/*Angenommen, die Argumentzeichenfolge lautet
"12,35.5,1"  */
ARG hours ',' rate ',' withhold
```

Das Muster wird aus der Analysezeichenfolge entfernt, wenn eine Übereinstimmung festgestellt wird. Wird die Analysezeichenfolge erneut von Anfang an abgesucht, kann die Zeichenfolge eine andere Länge und Struktur aufweisen als zu Beginn des Syntaxanalyseprozesses. Die ursprüngliche Quelle der Zeichenfolge wird jedoch niemals verändert.

7.3.3 Syntaxanalyse nach Positionsmarken

Die Syntaxanalyse mit Positionsmarken wird immer dann eingesetzt, wenn bekannt ist, daß sich die Felder von Interesse an bestimmten Positionen innerhalb einer Zeichenfolge befinden.

```
/* Sätze sehen wie folgt aus: */
/* Start:  1-5 */
/* Länge: 6-10 */
/* Name: fängt an bei (Start), hat (Lang)*/
PARSE value satz with 1 start +5 lang +5 =start name
+lang
```

Der zu verarbeitende Satz enthält ein variables Längenfeld. Startposition und Länge des Felds werden im ersten Teil des Satzes mit einer variablen Positionsmarke angegeben, aufgrund der das gewünschte Feld extrahiert wird.

Die Sequenz "=start" ist eine absolute Marke, deren Wert aus der Position hervorgeht, die weiter vorne im Suchvorgang in die Startvariable gestellt wurde. Die Sequenz "+lang" liefert die tatsächliche Länge des Felds.

7.3.4 Mehrere Schablonen

Zu einem Befehl können Sie auch mehrere Schablonen angeben. Dazu sind die einzelnen Schablonen jeweils durch ein Komma voneinander zu trennen. Der Befehl ARG (oder PARSE UPPER ARG) greift auf die Argumentzeichenfolgen zu, die beim Aufruf des Programms zur Verfügung gestellt wurden. Die Schablonen greifen nacheinander auf je eine Argumentzeichenfolge zu. Zum Beispiel:

```
/*Angenommen, die Argumente lauten ('eins
zwei,12,sort)*/
ARG erstes zweites,menge,aktion,option
```

Die erste Schablone besteht aus den Variablen erstes und zweites, die auf die Werte "eins" und "zwei" gesetzt werden. Die nächste Schablone weist den Wert "12" an "menge" zu, und "aktion" wird auf "SORT" gesetzt. Die letzte Schablone besteht aus der Variablen "option", die hier auf die leere Zeichenfolge gesetzt wird, da nur vier Argumente vorhanden waren.

Wenn mehrere Schablonen mit den Quellenoptionen EXTERNAL oder PULL verwendet werden, fordert jede weitere Schablone vom Benutzer eine weitere Eingabezeile an:

```
/*Lesen von nachname, vorname und weitere sowie von
datum*/
PULL nachname ',' vorname weitere,datum
```

Es werden zwei Eingabezeilen gelesen. Die erste Eingabezeile sollte drei Wörter enthalten, die den Variablen "nachname", "vorname" und "weitere" zugewiesen werden. Auf die erste Variable folgt ein Komma. Die gesamte zweite Eingabezeile wird der Variablen "datum" zugewiesen.

Mehrere Schablonen können auch mit einer Quellprogrammoption sinnvoll eingesetzt werden, die identische Zeichenfolgen zurückgibt. Enthielt die erste Schablone Mustermarken, die die Analysezeichenfolge veränderten, können die folgenden Schablonen noch immer auf die Originalzeichenfolge zugreifen. Später folgende Analysezeichenfolgen, die aus der Quelle VALUE hervorgehen, führen nicht dazu, daß der Ausdruck neu ausgewertet wird, sondern lediglich die jeweils vorherigen Resultate abgerufen werden.

Anhang A
Fehlermeldungen

Wenn der ARexx-Interpreter einen Fehler feststellt, gibt er einen Fehlercode zurück, der Aufschluß darüber gibt, um welche Art von Fehler es sich handelt. Normalerweise werden der Fehlercode, die Zeilennummer der Stelle im Quelltext, an der der Fehler auftrat, und eine kurze, den Fehler erläuternde Nachricht angezeigt. Anschließend wird die Verarbeitung des Programms abgebrochen und die Steuerung an die Aufrufebene zurückgegeben, es sei denn, es war der SYNTAX-Interrupt (über den Befehl SIGNAL) aktiviert. Die meisten Syntax- und Ausführungsfehler lassen sich mit Hilfe des SYNTAX-Interrupts abfangen. Dies gibt dem Programmierer die Gelegenheit, die geeigneten Maßnahmen zur Fehlerbehebung zu ergreifen. Manche Fehler werden aber auch außerhalb des Kontexts eines ARexx-Programms verursacht und können demzufolge nicht mit diesem Verfahren abgefangen werden. Nähere Informationen über das Abfangen und Bearbeiten von Fehlern finden Sie in Kapitel 6.

Jeder Fehlercode ist mit einem bestimmten Schweregrad verbunden, der dem aufrufenden Programm als primärer Ergebniscode gemeldet wird. Die Werte dieser Ergebniscodes sind 5 (geringfügiger Fehler), 10 (weniger schwerer Fehler) und 20 (sehr schwerer Fehler). Der Fehlercode selbst bildet das Sekundärergebnis. Die Art der Weitermeldung solcher Codes hängt vom jeweiligen externen (aufrufenden) Programm ab.

Auf den folgenden Seiten finden Sie eine Liste aller gegenwärtig definierten Fehlercodes mit den entsprechenden Ergebniscodes und dem jeweiligen Text der Fehlermeldung.

Tabelle A-1. Fehlercodes und Fehlermeldungen

Fehler	Fehler-code	Meldung	Erläuterung
1	5	Programm nicht gefunden (Program not found)	Das angegebene Programm wurde nicht gefunden oder war kein ARexx-Programm. Bei ARexx-Programmen wird vor-ausgesetzt, daß sie mit einem Kommentar (/*...*/) beginnen. Dieser Fehler wird von der externen Schnittstelle festge-stellt und kann nicht mit dem SYNTAX-Interrupt abgefangen werden.
2	10	Ausführung gestoppt (Execution halted)	Eine Unterbrechung durch Ctrl-C oder eine externe Halteanforderung wurde empfangen, worauf die Verarbeitung des Programms abgebrochen wurde. Dieser Fehler wird abgefangen, wenn der HALT-Interrupt aktiviert ist.
3	20	Speicher-platzmangel (Insufficient memory)	Der Interpreter konnte nicht genügend Speicherplatz für eine bestimmte Operation reservieren. Speicherplatz wird für alle Analyse- und Ausführungsoperationen benötigt. Der Fehler kann daher normalerweise nicht mit dem SYNTAX-Interrupt abgefangen werden.

Fehler	Fehler-code	Meldung	Erläuterung
4	10	Ungültiges Zeichen (Invalid character)	Im Programm wurde ein Nicht-ASCII-Zeichen (z. B. ein Umlaut in einem Variablen-namen) entdeckt. Steuercodes und andere Nicht-ASCII-Zeichen können in einem Programm verwendet werden. Dazu müssen sie aber als Hexadezimal- oder Binär-zeichenfolgen definiert werden. Dies ist ein Suchphasenfehler und kann nicht mit dem SYNTAX-Interrupt abgefangen werden.
5	10	Ungerade Anzahl von Anführungs-zeichen (Unmatched quote)	Es fehlt ein abschließendes einfaches oder doppeltes Anführungszeichen. Über-prüfen Sie das Programm auf die korrekte Begrenzung aller Zeichenfolgen. Dies ist ein Suchphasenfehler und kann nicht mit dem SYNTAX-Interrupt abgefangen werden.
6	10	Fehlendes Kommentar-abschluß-zeichen (Unterminated comment)	Das Abschlußzeichen */ eines Kommentars wurde nicht gefunden. Beachten Sie, daß Kommentare auch verschachtelt sein können. Zu jedem /* muß ein */ vorhanden sein. Dies ist ein Suchphasenfehler und kann nicht mit dem SYNTAX-Interrupt abgefangen werden.

Fehler	Fehler-code	Meldung	Erläuterung
7	10	Klausel zu lang (Clause too long)	Eine Klausel war für den internen Puffer zu lang. Die Quelltextzeile sollte in kleinere Komponenten aufgeteilt werden. Dies ist ein Suchphasenfehler und kann nicht mit dem SYNTAX-Interrupt abgefangen werden.
8	10	Ungültiges Token (Invalid token)	Ein nicht erkanntes lexikalisches Token wurde entdeckt, oder eine Klausel konnte nicht korrekt klassifiziert werden. Dies ist ein Suchphasenfehler und kann nicht mit dem SYNTAX-Interrupt abgefangen werden.
9	10	Symbol oder Zeichenkette zu lang (Symbol or string too long)	Es wurde versucht, eine Zeichenkette zu erstellen, die die maximal zulässige Länge überschritt.
10	10	Ungültiges Nachrichtenpaket (Invalid message packet)	In einem Nachrichtenpaket, das an den residenten ARexx-Prozeß gesendet wurde, wurde ein ungültiger Aktionscode entdeckt. Das Paket wurde unverarbeitet zurückgegeben. Dieser Fehler wird von der externen Schnittstelle festgestellt und kann nicht mit dem SYNTAX-Interrupt abgefangen werden.

Fehler	Fehler-code	Meldung	Erläuterung
11	10	Fehler in der Kommando-zeichenfolge (Command string error)	Eine Kommandozeichenfolge konnte nicht verarbeitet werden. Dieser Fehler wird von der externen Schnittstelle fest-gestellt und kann nicht mit dem SYNTAX-Interrupt abgefangen werden.
12	10	Fehlerrück-gabe von der Funktion (Error return from function)	Eine externe Funktion gab einen von Null abweichenden Fehlercode zurück. Stellen Sie sicher, daß zu der Funktion korrekte Argumente angege-ben wurden.
13	10	Host-Umgebung nicht gefunden (Host environment not found)	Der einer Host-Adreßzeichen-folge entsprechende Message-Port wurde nicht gefunden. Stellen Sie sicher, daß der betreffende externe Host aktiv ist.
14	10	Angeforderte Bibliothek nicht gefunden (Requested library not found)	Es wurde versucht, eine in der Bibliotheksliste enthaltene Funktionsbibliothek zu öffnen, die Bibliothek konnte jedoch nicht geöffnet werden. Stellen Sie sicher, daß Name und Version der Bibliothek korrekt angegeben wurden, als die Bibliothek in die Ressourcen-liste aufgenommen wurde.

Fehler	Fehler-code	Meldung	Erläuterung
15	10	Funktion nicht gefunden (Function not found)	Es wurde eine Funktion gefunden, die weder in einer der gegenwärtig im Zugriff befindlichen Bibliotheken gefunden noch als externes Programm lokalisiert werden konnte. Stellen Sie sicher, daß die entsprechenden Funktions-bibliotheken in die Biblio-thekenliste aufgenommen wurden.
16	10	Funktion gab keinen Wert zurück (Function did not return value)	Es wurde eine Funktion auf-gerufen, die keine Ergebnis-zeichenfolge zurückgab, aber auch sonst keinen Fehler meldete. Stellen Sie sicher, daß die Funktion korrekt programmiert wurde oder rufen Sie die Funktion über den Befehl CALL auf.
17	10	Falsche Anzahl an Argumenten (Wrong number of arguments)	Es wurde eine Funktion aufgerufen, die eine andere Anzahl Argumente erwartet. Dieser Fehler wird generiert, wenn eine integrierte oder externe Funktion mit mehr Argumenten aufgerufen wird, als das Nachrichtenpaket für die externe Kommunikation aufnehmen kann.
18	10	Ungültiges Argument für Funktion (Invalid argument to function)	Einer Funktion wurde ein unpassendes Argument mitgegeben, oder es fehlte ein erforderliches Argument. Überprüfen Sie die erforder-lichen Argumente der betreffenden Funktion.

Fehler	Fehlercode	Meldung	Erläuterung
19	10	Ungültige PROCEDURE (Invalid PROCEDURE)	Ein PROCEDURE-Befehl wurde in einem ungültigen Kontext gegeben. Entweder waren keine internen Funktionen aktiv, oder ein PROCEDURE-Befehl war zuvor bereits in der aktuellen Speicherumgebung gegeben worden.
20	10	Unerwartetes THEN oder WHEN (Unexpected THEN or WHEN)	Ein WHEN- oder THEN-Befehl wurde außerhalb eines gültigen Kontexts aufgerufen. Der Befehl WHEN ist nur innerhalb eines SELECT-Bereichs gültig, THEN muß als nächster Befehl auf IF oder WHEN folgen.
21	10	Unerwartetes ELSE oder OTHERWISE (Unexpected ELSE or OTHERWISE)	Ein ELSE- oder OTHERWISE-Befehl wurde außerhalb eines gültigen Kontexts gefunden. Der Befehl OTHERWISE ist nur innerhalb eines SELECT-Bereichs gültig, ELSE ist nur als Folge eines THEN-Zweigs innerhalb eines IF-Bereichs gültig.
22	10	Unerwartetes BREAK, LEAVE oder ITERATE (Unexpected BREAK, LEAVE or ITERATE)	Der Befehl BREAK ist nur innerhalb eines DO-Bereichs oder innerhalb einer INTERPRETierten Zeichenfolge gültig. Die Befehle LEAVE und ITERATE sind nur innerhalb eines iterativen DO-Bereichs gültig.

Fehler	Fehler-code	Meldung	Erläuterung
23	10	Ungültige Anweisung in SELECT (Invalid statement in SELECT)	Innerhalb eines SELECT-Bereichs wurde eine ungültige Anweisung entdeckt. Nur die Anweisungen WHEN, THEN und OTHERWISE sind innerhalb eines SELECT-Bereichs gültig, abgesehen von den bedingten Anweisungen, die auf eine der Klauseln THEN oder OTHERWISE folgen können.
24	10	Fehlendes oder mehrfach vorkommendes THEN (Missing or multiple THEN)	Eine vom Programm erwartete THEN-Klausel wurde nicht gefunden, oder es wurde ein weiteres THEN festgestellt, nachdem bereits eines ausgeführt wurde.
25	10	Fehlendes OTHERWISE (Missing OTHERWISE)	Keine der WHEN-Klauseln in einem SELECT-Bereich war erfolgreich, dennoch war keine OTHERWISE-Klausel vorhanden.
26	10	Fehlendes oder unerwartetes END (Missing or unexpected END)	Der Quelltext des Programms war beendet, bevor ein DO- oder SELECT-Befehl mit END abgeschlossen war, oder ein END-Befehl wurde außerhalb eines DO- oder SELECT-Bereichs entdeckt.

Fehler	Fehler-code	Meldung	Erläuterung
27	10	Symbol ohne Entspre-chung (Symbol mismatch)	Das in einem END-, ITERATE- oder LEAVE-Befehl angege-bene Symbol entsprach nicht der Indexvariablen des DO-Bereichs. Stellen Sie sicher, daß die aktiven Schleifen korrekt verschachtelt sind.
28	10	Ungültige DO-Syntax (Invalid DO syntax)	Ein ungültiger DO-Befehl wurde aufgerufen. Bei Angabe eines Ausdrucks nach TO oder BY ist auch ein Initialisierungs-ausdruck erforderlich. Ein FOR-Ausdruck muß zu einem ganzzahligen, nicht negativen Ergebnis führen.
29	10	Unvollstän-diges IF oder SELECT (Incomplete IF or SELECT)	Ein IF- oder SELECT-Bereich endete, bevor alle dafür erfor-derlichen Anweisungen ge-funden werden konnten. Über-prüfen Sie, ob die bedingte An-weisung, die auf eine THEN-, ELSE- oder OTHERWISE-Klausel folgen muß, möglicher-weise fehlt.
30	10	Sprung-marke nicht gefunden (Label not found)	Eine Sprungmarke, die in einem SIGNAL-Befehl ange-geben oder auf die durch einen aktiven Interrupt implizit verwiesen wurde, konnte im Quelltext des Programms nicht gefunden werden. Sprung-marken, die durch einen INTERPRET-Befehl oder eine Dialogeingabe dynamisch aktiviert wurden, werden vom Suchvorgang nicht abgedeckt.

Fehler	Fehler-code	Meldung	Erläuterung
31	10	Symbol wurde erwartet (Symbol expected)	Ein Token, das kein Symbol ist, wurde an einer Stelle ange-troffen, an der nur ein Symbol-Token zulässig ist. Auf die Befehle DROP, END, LEAVE, ITERATE und UPPER darf nur ein Symbol-Token folgen. Diese Meldung wird auch aus-gegeben, wenn ein erforder-liches Symbol fehlt.
32	10	Symbol oder Zeichenfolge wurde erwartet (Symbol or string expected)	Ein ungültiges Token wurde in einem Kontext angetroffen, in dem nur ein Symbol oder eine Zeichenfolge zulässig ist.
33	10	Ungültiges Schlüssel-wort (Invalid keyword)	Ein Symbol-Token in einer Befehlsklausel wurde als Schlüsselwort identifiziert, war aber im betreffenden Kontext ungültig.
34	10	Erfoderliches Schlüssel-wort fehlt (Required keyword missing)	Eine Befehlsklausel erfordert ein bestimmtes Schlüsselwort-Token, dieses war aber nicht vorhanden. Diese Meldung wird ausgegeben, wenn auf einen Befehl SIGNAL ON kein Interrupt-Schlüsselwort folgt (z. B. SYNTAX).
35	10	Nicht zugehörige Zeichen (Extraneous characters)	Eine scheinbar gültige An-weisung wurde ausgeführt, am Ende der Klausel wurden jedoch zusätzliche Zeichen festgestellt.

Fehler	Fehler-code	Meldung	Erläuterung
36	10	Schlüssel-wortkonflikt (Keyword conflict)	Eine Befehlsklausel enthielt zwei Schlüsselwörter, die sich gegenseitig ausschließen, oder ein Schlüsselwort wurde zweimal im gleichen Befehl angegeben.
37	10	Ungültige Schablone (Invalid template)	Die zu einem ARG-, PARSE- oder PULL-Befehl angegebene Schablone war nicht korrekt aufgebaut.
38	10	Ungültige TRACE-Anforderung (Invalid TRACE request)	Das zu einem TRACE-Befehl oder als Argument zu einer integrierten TRACE()-Funktion angegebene alphabetische Schlüsselwort war ungültig.
39	10	Nicht initialisierte Variable (Uninitialized variable)	Es wurde versucht, eine nicht initialisierte Variable zu verwenden, während der NOVALUE-Interrupt aktiviert war.
40	10	Ungültiger Variablen-name (Invalid variable name)	Es wurde versucht, einem festen Symbol einen Wert zuzuordnen.

Fehler	Fehler-code	Meldung	Erläuterung
41	10	Ungültiger Ausdruck (Invalid expression)	Bei der Bewertung eines Ausdrucks wurde ein Fehler festgestellt. Stellen Sie sicher, daß zu jedem Operator die korrekte Anzahl Operanden angegeben wurde und daß keine nicht zugehörigen Zeichen im Ausdruck vorkommen. Dieser Fehler wird nur in Ausdrücken festgestellt, die tatsächlich ausgewertet werden. Bei Ausdrücken in Klauseln, die übersprungen werden, erfolgt keine Überprüfung.
42	10	Fehlerhafte Anzahl runder Klammern (Unbalanced parentheses)	Ein Ausdruck wurde entdeckt, in dem die Anzahl der öffnenden Klammern von der Anzahl der schließenden Klammern abweicht.
43	10	Verschachtelungsgrenzwert überschritten (Nesting limit exceeded)	Die Anzahl der Unterausdrücke innerhalb eines Ausdrucks überschreitet das zulässige Maximum. Vereinfachen Sie den Ausdruck, indem Sie ihn in zwei oder mehr kleinere Ausdrücke zerlegen.
44	10	Ungültiges Ergebnis eines Ausdrucks (Invalid expression result)	Das Ergebnis eines Ausdrucks war innerhalb seines Kontexts ungültig. Diese Meldung wird ausgegeben, wenn ein Inkrement oder Grenzwertausdruck in einem DO-Befehl zu einem nicht numerischen Ergebnis führt.

Fehler	Fehler-code	Meldung	Erläuterung
45	10	Ausdruck erforderlich (Expression required)	Ein Ausdruck wurde in einem Kontext weggelassen, in dem er erforderlich gewesen wäre. Auf den Befehl SIGNAL muß z. B. ein Ausdruck folgen, es sei denn, es folgt eines der Schlüsselwörter ON oder OFF.
46	10	Boolescher Wert ist nicht 0 oder 1 (Boolean value not 0 or 1)	Als Ergebnis eines Ausdrucks wurde ein boolescher Wert erwartet, die Auswertung ergab aber etwas anderes als 0 oder 1.
47	10	Fehlerhafte arithmeti-sche Konver-sion (Arithmetic conversion error)	Ein nicht numerischer Operand wurde in einer Operation ver-wendet, die numerische Operanden verlangt. Diese Meldung wird auch im Falle einer ungültigen hexadezi-malen oder binären Zeichen-folge generiert.
48	10	Ungültiger Operand (Invalid operand)	Ein Operand war für die beabsichtigte Operation nicht gültig. Diese Meldung wird generiert, wenn versucht wurde, durch 0 zu dividieren oder einen Bruch als Exponen-ten in einer Potenzierungs-operation zu verwenden.

Anhang B
Kommandohilfsprogramme

ARexx bietet eine Reihe von Kommandohilfsprogrammen mit verschiedenen Steuerfunktionen. Diese Programme stehen im Verzeichnis REXXC. Dabei handelt es sich um ausführbare Programme, die von der Shell aus aufgerufen werden können und nur dann relevant sind, wenn der residente ARexx-Prozeß aktiv ist.

HI (Halt-Interrupt) HI

Setzt das globale Halt-Kennzeichen, das bewirkt, daß alle laufenden ARexx-Programme eine externe Halt-Anforderung empfangen. Jedes Programm wird augenblicklich verlassen, es sei denn, für das Programm ist der HALT-Interrupt aktiviert. Das Halt-Kennzeichen bleibt nicht gesetzt, sondern wird automatisch gelöscht, nachdem alle laufenden Programme die Halt-Anforderung erhalten haben.

RX (Rexx eXecute) RX Name [Argumente]

Startet ein ARexx-Programm. Falls der angegebene Name eine explizite Pfadangabe enthält, wird nur in diesem Verzeichnis nach dem Programm gesucht. Andernfalls werden das aktuelle Verzeichniss sowie REXX: nach einem Programm des angegebenen Namens durchsucht. Die wahlfrei anzugebende Argumentzeichenfolge wird an das Programm weitergeleitet.

RXSET RXSET [Name [[=] Wert]]

Fügt ein (Name,Wert)-Paar in die Clip-Liste ein. Bei Namenszeichenfolgen wird vorausgesetzt, daß sie Groß- und Kleinschreibung enthalten. Wenn ein Paar dieses Namens bereits existiert, wird sein Wert durch die aktuelle Zeichenfolge ersetzt. Wird ein Name ohne Wertzeichenfolge angegeben, so wird der Eintrag aus der Clip-Liste entfernt. Wird RXSET ohne Argumente aufgerufen, werden alle (Name, Wert)-Paare aus der Clip-Liste aufgelistet.

RXC (REXX Close) RXC

Schließt den residenten Prozeß. Der allgemein zugängliche "REXX"-Port wird unverzüglich geschlossen. Der residente Prozeß wird verlassen, sobald das letzte ARexx-Programm abgeschlossen ist. Nach einer solchen Abschlußanforderung können keine weiteren Programme gestartet werden.

TCC (TraCe Close) TCC

Schließt die globale Ablaufverfolgungskonsole, sobald sie von keinem aktiven Programm mehr benutzt wird. Alle an der Konsole anstehenden Leseanforderungen müssen beantwortet worden sein, bevor sie geschlossen werden kann.

TCO (TraCe Open) TCO

Öffnet die globale Ablaufverfolgungskonsole. Die Ausgabedaten der Ablaufverfolgung aller aktiven Programme werden automatisch an die neue Konsole umgeleitet. Der Benutzer kann Größe und Position des Konsolenfensters verändern, und er kann das Konsolenfenster mit dem Kommando TCC schließen.

TE (Trace End) TE

Löscht das Kennzeichen für globale Ablaufverfolgung. Damit wird der Ablaufverfolgungsmodus für alle aktiven ARexx-Programme auf OFF gesetzt.

TS (Trace Start) TS

Startet die interaktive Ablaufverfolgung, indem das externe Ablaufverfolgungskennzeichen gesetzt wird. Damit werden alle aktiven ARexx-Programme in den interaktiven Ablaufverfolgungsmodus gebracht. Die Programme beginnen, Ausgabedaten zur Ablaufverfolgung zu erzeugen, und pausieren nach der nächsten Anweisung. Dieser Befehl ist nützlich, wenn Sie die Kontrolle über Programme wiedererlangen wollen, die sich in Endlosschleifen befinden oder andere ungewöhnliche Fehlersymptome zeigen. Das Ablaufverfolgungskennzeichen bleibt solange gesetzt, bis es mit dem Kommando TE gelöscht wird, d. h. später aufgerufene Programme werden ebenfalls im interaktiven Ablaufverfolgungsmodus ausgeführt.

WaitForPort WaitForPort [Name des Ports]

WaitForPort wartet 10 Sekunden, bis der angegebene Port bereitgestellt wird. Der Rückgabewert 0 bedeutet, daß der Port gefunden wurde. Der Rückgabewert 5 zeigt an, daß die Anwendung gegenwärtig nicht läuft oder daß der Port nicht existiert. Bei Port-Namen wird zwischen Groß- und Kleinschreibung unterschieden. Beispiel:

```
WaitForPort ED_1
WaitForPort MeinPort
```

Glossar

In diesem Glossar finden Sie Definitionen von Fachbegriffen, die im ARexx-Handbuch verwendet werden.

Ablaufverfolgung
Zeigt die Zeilen eines ARexx-Programms an, während dieses ausgeführt wird. Auf dieses Weise lassen sich Fehler exakt lokalisieren.

Adresse
Eine Kennummer, die jedem Datenbyte in einem Computerspeicher und jedem Sektor auf einer Festplatte oder Diskette zugeordnet wird.

Anweisung
Eine Zuweisungs-, Befehls- oder Kommandoklausel.

Argument
Eine Zusatzinformation zu einem Befehl oder einer Funktion. Das Argument bestimmt die auszuführende Aktion.

Ausdruck
Eine Gruppe auszuwertender Tokens. Ausdrücke bestehen aus Zeichenfolgen, Symbolen, Operatoren und runden Klammern.

Befehl
Eine Klausel, die mit einem bestimmten Schlüsselwort beginnt und ARexx zur Ausführung einer Aktion auffordert.

Begrenzungszeichen
Ein Zeichen, das Anfang und Ende einer Zeichenfolge markiert. ARexx-Begrenzungszeichen sind das einfache (') und das doppelte Anführungszeichen (").

Bibliotheksliste
Eine interne, von ARexx verwaltete Liste aller gegenwärtig verfügbaren Funktionsbibliotheken und Funktions-Hosts. Anwendungen können bei Bedarf Funktionsbibliotheken hinzufügen oder entfernen.

Boolesch
Werte mit zwei möglichen Zuständen: 0 (falsch) oder 1 (wahr).

Clip-Liste
Ein Zwischenspeicher, der für die Kommunikation zwischen Tasks gebraucht wird. Mit der Funktion SETCLIP() können Begriffspaare (Name, Wert) in die Clip-Liste aufgenommen werden.

Einfaches Symbol
Ein Token, das nicht mit einer Ziffer beginnt und keine Punkte enthält.

Fehlersuche (Debugging)
Das Lokalisieren und Beheben von Programmfehlern.

Festes Symbol
Ein Token, das mit einer Ziffer oder einem Punkt beginnt.

Funktion
Eine Gruppe von Anweisungen, die als zusammengehöriger Block ausgeführt werden kann. Funktionen ermöglichen den Aufbau komplexer Programme aus kleineren Modulen.

Funktions-Host
Eine externe Anwendung mit ARexx-Port. Der Name eines Funktions-Hosts ist der Name des allgemein zugänglichen Message-Ports des Programms.

Funktionsbibliotheken
Eine Sammlung einer oder mehrerer Funktionen, die als gemeinsam benutzbare Amiga-Bibliothek aufgebaut ist. Eine Funktionsbibliothek muß einen Bibliotheksnamen, eine Suchpriorität, einen Offset für die Einstiegsposition und eine Versionsnummer enthalten.

Globale Umgebung
Die festen Komponenten eines ARexx-Programms. Dazu gehören der Quellencode des Programms, statische Datenzeichenfolgen und Argumentzeichenfolgen.

Host-Adresse
Innerhalb und außerhalb einer Anwendung ist die Host-Adresse der Name des Message-Ports, an den ARexx-Kommandos gesendet werden können.

Interrupts
Interne Kennzeichen in ARexx, mit deren Hilfe ein Programm Fehler feststellen und die Steuerung in Fällen beibehalten kann, in denen das Programm ohne solche Kennzeichen abgebrochen würde. Interrupts werden normalerweise über den Befehl SIGNAL: gesteuert.

Iteration
Wiederholen eines Programmabschnitts.

Klausel
Die kleinste ausführbare Einheit einer Programmiersprache.

Kommandoklausel
Ein ARexx-Ausdruck, in welchem das Ergebnis als Kommando an eine externe Anwendung ausgegeben wird.

Kommandoschnittstelle
Ein Message-Port, über den ARexx-Kommandos an kompatible Anwendungen ausgegeben werden.

Kommentar
Eine Gruppe von Zeichen, die zwischen den Symbolen /*....*/ steht. Jedes ARexx-Programm beginnt mit einem Kommentar.

Makro
Andere Bezeichnung für ein ARexx-Programm.

Markierung
Ein Token, das Anfang und Ende einer Analysezeichenfolge bestimmt.

Message-Port
Eine Schnittstelle in einer Amiga-Anwendung, über die das Programm mit einem ARexx-Programm kommunizieren kann.

Nullklausel
Eine Leerzeile oder Kommentarzeile.

Numerische Genauigkeit
Die Anzahl der Dezimalstellen in einem arithmetischen Ergebnis. Je weniger Dezimalstellen, desto ungenauer das Ergebnis.

Operatoren
Ein Zeichen wie (+), (–) oder (|), das bei arithmetischen Verkettungen, Vergleichen oder logischen Operationen verwendet wird.

Prozeßkommunikation
Datenaustausch zwischen Anwendungen.

RexxMast
Das Programm, das als Interpreter für ARexx-Programme eingesetzt wird.

Rückgabewert
Der Schweregrad eines Fehlers. Diese Zahl (5, 10 oder 20) wird angezeigt, wenn es in einem ARexx-Programm zu einem Fehler kommt.

Schablone
Eine Gruppe von Tokens, die angibt, welche Variablen in einer Syntaxanalyseoperation zu verwenden sind. Die Schablone bestimmt auch die Art und Weise, in der die Werte festgelegt werden.

Speicherumgebung
Die variablen Komponenten eines ARexx-Programms. Dazu gehören die Symboltabelle, numerische Optionen, Ablaufverfolgungsoptionen und Host-Adreßzeichenfolgen.

Sprungmarke
Ein Symbol-Token, gefolgt von einem Doppelpunkt (:). Sprungmarken bezeichnen bestimmte Positionen in einem Programm.

Stammsymbol
Ein Token mit einem Punkt am Ende seines Namens. Stammsymbole dienen zur Initialisierung zusammengesetzter Symbole.

Symbol
Eine beliebige Gruppe in der die alphanumerischen Zeichen a-z, A-Z, 0-9, Punkt (.), Ausrufezeichen (!), Fragezeichen (?), Dollarzeichen ($) und Unterstreichungszeichen (_) vorkommen können.

Symboltabelle
Eine interne, von ARexx erstellte Tabelle, in der Wertzeichenfolgen gespeichert werden, die den Variablen in einem Programm zugeordnet wurden.

Syntaxanalyse
Das Zerlegen einer Zeichenfolge in kleinere Einheiten.

Token
Die kleinsten Einheiten der Programmiersprache ARexx.

Tokenisierung
Das Zerlegen einer Anweisung in ihre einzelnen Tokens.

Variable
Ein Symbol, dem ein Wert zugeordnet werden kann.

Verkettung
Der Vorgang der Verknüpfung (Aneinanderreihung) zweier Zeichenfolgen.

Zeichenfolge
Eine Gruppe von Zeichen, die mit einem Begrenzungszeichen beginnt und endet (einfache oder doppelte Anführungszeichen). Der Wert einer Zeichenfolge ist die Zeichenfolge selbst.

Ziel
Ein normalerweise variables Symbol, das im Verlauf einer Syntaxanalyse einem Wert zugeordnet wird.

Zuordnungsklausel

Ein variables Symbol (einfach, zusammengesetzt oder Stammsymbol), auf das der Operator = folgt. In einer Zuordnungsklausel, werden die Tokens rechts vom Gleichheitszeichen ausgewertet und das Resultat dem variablen Symbol zugeordnet.

Zusammengesetztes Symbol

Ein Token, in dem alphanumerische Zeichen, die Zeichen ., !, ?, $ und _ sowie ein oder mehrere Punkte innerhalb eines Namens vorkommen können. Zusammengesetzte Symbole besitzen die Struktur Stamm.n1.n2...nk.

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Z

Notizen

Notizen